KARELIA-AMMATTIKORKEAKOULU
Tietojenkäsittelyn koulutus

Lassi Peltola

OHJELMISTOTESTAUS NODE.JS & REACT.JS - KEHITYSYMPÄRISTÖISSÄ

Opinnäytetyö
Helmikuu 2020

Karelia AMMATTIKORKEAKOULU

OPINNÄYTETYÖ
Tammikuu 2020
Tietojenkäsittelyn koulutus

Tikkarinne 9
80200 JOENSUU
+358 13 260 600 (vaihde)

Tekijä
Lassi Peltola

Nimeke
Ohjelmistotestaus Node.js & React.js -kehitysympäristöissä

Toimeksiantaja
Solenovo Oy

Tiivistelmä

Tässä opinnäytetyössä perehdyttiin ohjelmistotestauksen perusteisiin sekä tutkittiin Node.js ja React.js -kehityksessä käytettäviä testauskehyksiä ja -työkaluja. Opinnäytetyön toimeksiantajana toimi ohjelmistotalo Solenovo Oy, joka toteuttaa järjestelmiä oppilaitoksille ja julkishallinnolle. Opinnäytetyön tarkoitus oli tutkia Javascript-testausta ja tuoda toimeksiantajalle lisää tietoa Node.js ja React.js -testauksesta sekä siirtää testausosaamista alihankkijalta toimeksiantajalle. Lisäksi työssä oli tarkoitus mitata ja pohtia, miten työn aikana tutkittu tieto vaikuttaa yrityksen testauskäytäntöihin.

Opinnäytetyön toiminnallisessa osiossa testejä toteutettiin toimeksiantajan työaikapankkiprojektissa. Node.js -osiossa testattiin rest-rajapintaa käyttäen Mocha-, Jest- ja vREST -työkaluja. React.js -osiossa testattiin käyttöliittymää käyttäen Selenium IDE-, Screener E2E sekä Cypress -testaustyökaluja.

Työn tuloksena saatiin kerättyä ja dokumentoitua tietoa erilaisista Javascript-kehityksessä käytettävistä testaustyökaluista. Testausosaamista saatiin siirrettyä alihankkijalta toimeksiantajalle. Opinnäytetyön myötä kerättyä uutta tietoa on tulevaisuudessa tarkoitus jakaa yrityksessä ja tutkittujen testaustyökalujen käyttöä tullaan kouluttamaan kehittäjien kesken.

Kieli
suomi

Sivuja 73

Asiasanat
ohjelmistotestaus, javascript, node.js, react.js

Karelia AMMATTIKORKEAKOULU

THESIS
January 2020
Business Information Technology

Tikkarinne 9
80200 JOENSUU
+358 13 260 600 (switchboard)

Author
Lassi Peltola

Title
Software Testing in Node.js & React.js Development Environment

Commissioned by
Solenovo Oy

Abstract

The purpose of this thesis was to become familiar with the basics of software testing and study testing tools that are used for Node.js and React.js development. This thesis was commissioned by Solenovo Ltd, a software company which develops systems for school institutions and public administrations. The thesis aimed to study Javascript-testing to provide the commissioner with more information about Node.js and React.js testing and transfer existing testing knowledge from subcontractor to commissioner. In addition, this thesis intended to measure and reflect how the new researched information will affect commissioners testing practices.

In the functional part of this thesis tests were implemented in the commissioners working time bank project. In the Node.js section, a rest interface was tested with Mocha, Jest and vREST. In the React.js section, the applications user interface was tested with Selenium IDE, Screener E2E and Cypress.

As a result of this thesis, information about various Javascript testing tools was gathered and documented. The testing knowledge was successfully transferred from subcontractor to commissioner. Information gathered for this thesis will be shared with Solenovo Ltd and the usage of the testing tools will be introduced to the developers of the company.

Language
Finnish

Pages 73

Keywords
software testing, javascript, node.js, react.js

Sisältö

1 Johdanto

Tämä opinnäytetyö toteutetaan toimeksiantona Solenovo Oy:lle. Solenovo on vuonna 1996 perustettu joensuulainen ohjelmistoalan yritys, joka toteuttaa toiminnanohjaus- ja projektinhallintajärjestelmiä oppilaitoksille sekä julkiselle sektorille. Opinnäyteyössä on tarkoitus perehdyttää lukija ohjelmistotestauksen perusteisiin sekä tutkia modernien Javascript-kirjastojen testaukseen käytettäviä testauskehyksiä ja kehittää testejä toimeksiantajan uudessa projektissa. Toimeksiantajan muissa projekteissa käytössä olevien testausteknologioiden osaamista on tarkoitus siirtää alihankkijalta toimeksiantajalle. Opinnäytetyön toiminnallisessa osiossa on tavoitteena tutkia erilaisten verkkosovelluksissa käytettävien testaustyökalujen käyttöä. Miten rest-rajapintaa testataan Mocha-, Jest- ja vREST -työkaluilla? Kuinka React-käyttöliittymää testataan Selenium IDE-, Screener E2E ja Cypress -työkaluilla? Lopuksi toteutettujen testien ja opinnäytetyön tuoman testaustietoisuuden vaikutuksia on tarkoitus mitata ja pohtia. Miten tutkittu tieto vaikuttaa yrityksen testauskäytäntöihin tulevaisuudessa? Onnistuiko testausosaamisen siirtäminen alihankkijalta toimeksiantajalle?

Opinnäytetyö koostuu kolmesta osiosta. Ensimmäisessä osiossa käsitellään ohjelmistotestausta hieman yleisemmällä ja teoreettisemmalla tasolla. Osion tarkoitus on antaa lukijalle käsitys siitä, mistä ohjelmistotestauksessa on kyse sekä miksi ja miten ohjelmistoja testataan. Toisessa osiossa perehdytään testaukseen Javascript-kehityksessä ja kerrotaan tarkemmin testattavasta projektista sekä työssä käytettävistä teknologioista ja testauskehyksistä. Lisäksi siinä on käytännön osio, jossa testejä toteutetaan toimeksiantajan työaikapankki-sovelluksessa valittuja testauskehyksiä hyödyntäen. Kolmannessa osiossa käsitellään opinnäytetyön tuloksia ja analysoidaan sitä, kuinka opinnäytetyön tuoma lisätieto Javascript-testauksesta on vaikuttanut yrityksen tuotteiden laatuun ja tapaan kehittää ohjelmistoja sekä miten opinnäytetyön tulokset vastaavat sille asetettuihin tavoitteisiin.

2 Ohjelmistotestauksen perusteet

2.1 Miksi ohjelmistoja testataan?

Terminä ohjelmistotestauksella tarkoitetaan prosessia, jonka tarkoitus on varmistaa, että toteutettava tuote on vaatimusten mukainen ja että kaikki sen ominaisuudet toimivat halutulla tavalla. Yritykset käyttävät eri teollisuudenaloilla 25–65 prosenttia projektien kokonaisbudjetista testaukseen, mikä tekee testauksesta ohjelmistoprojektien kalleimman yksittäisen kokonaisuuden. (Kasurinen 2013, 10–12.)

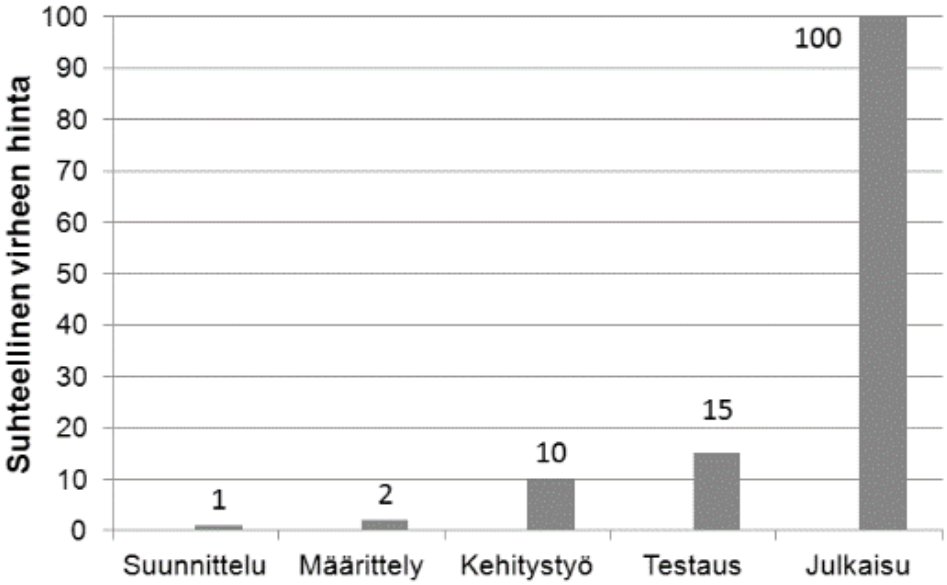
Ohjelmistot ovat usein laajoja kokonaisuuksia, jotka koostuvat monista pienemmistä osista. Erittäin pienissä projekteissa testaukselle ei välttämättä olekaan tarvetta, kun kehittäjä pystyy muistamaan ja ymmärtämään koko ohjelmakoodin. Hyvin nopeasti järjestelmän ja koodikannan laajentuessa yksi kehittäjä tai edes koko kehitystiimi ei kuitenkaan pysty enää seuraamaan, mihin kaikkeen muutokset koodissa voivat vaikuttaa. (Vala Group 2019.) Myös tiimin kokoonpano voi muuttua, jolloin tiimistä poistuvan kehittäjän pään sisällä ollut dokumentoimaton tieto järjestelmän toiminnasta katoaa käytännössä olemattomiin. Kun järjestelmään ei ole tehty testejä riittävällä kattavuudella, voi uusien ominaisuuksien lisääminen aiheuttaa järjestelmässä häiriön tai jopa sen rikkoutumisen. Ohjelmiston häiriö saa alkunsa siitä, kun kehittäjä tekee ohjelmakoodia kirjoittaessaan erehdyksen, joka johtaa vikaan eli bugiin järjestelmässä. Kun järjestelmä suorituksen aikana kohtaa bugin, seuraa siitä virhetoiminto, joka voi johtaa järjestelmän kaatumiseen tai ei-odotetun arvon palauttamiseen. (Tuovinen 2013, 14.)

Ohjelmistojen viat ovat johtaneet suuriin rahallisiin tappioihin yrityksissä, ja ne ovat pahimmissa tapauksissa vaatineet satoja ihmishenkiä katastrofaalisissa onnettomuuksissa. Esimerkiksi ESA:n miehittämätön Ariane 5 -kantoraketti tuhoutui vuonna 1996 ohjelmointivirheen vuoksi, kun raketin inertiaa säätelevä ohjelmisto yritti muuttaa raketin nopeuden 64-bittisestä luvusta 16-bittiseksi luvuksi nopeuden ollessa suurempi kuin muuttujan suurin sallittu arvo. Rakettia kehitettiin kymmenen vuotta, ja sen tuhoutuminen aiheutti seitsemän miljardin dollarin tappiot.

(Arnold 2000.) Lokakuussa 2018 ja maaliskuussa 2019, vain viiden kuukauden sisällä toisistaan, kaksi Boeing 737 Max -lentokonetta syöksyi maahan vaatien yhteensä 346 ihmishenkeä. Onnettomuuden takana oli useita virheitä ja huonoja päätöksiä, mutta yksi onnettomuuteen johtaneista tekijöistä oli lentokoneiden MCAS-vakausjärjestelmässä ollut vika. (Gelles 2019.)

On siis erittäin tärkeää, että tuotantoon pääsevät ohjelmistot ovat riittävän kattavasti testattuja. Testauksen tärkein tavoite onkin löytää ohjelmistosta viat, korjata ne ja todeta korjatut kohdat toimiviksi ennen kuin tuotteen voi ottaa käyttöön. Kun testauksen tavoitteet on suunnitteluvaiheessa määritelty, voidaan testauksella myös todistaa tuotteen laatu. Laadukas tuote täyttää sille asetetut vaatimukset asiakkaan sekä yleisten asioiden, kuten lakien ja standardien osalta. (Tieturi 2006, 5.)

Vikojen ennaltaehkäisy on etenkin kustannussyistä kannattavaa, sillä mitä pidemmälle ohjelmistoprojekti etenee, sitä kalliimmaksi vikojen korjaus tulee. Jussi Pekka Kasurinen (2013, 17–18) kertookin Ohjelmistotestauksen käsikirjassa, että suunniteltaessa löydetty virhe maksaa kymmenesosan kehitystyön aikana havaitusta virheestä ja sadasosan julkaisun aikana löydetystä virheestä (kuvio 1). Kyseessä on 1960-luvulta saakka yleisesti käytetty tuotteen taloudellista laatua kuvaava metriikka, joka on saanut viime aikoina myös kritiikkiä. Amerikkalainen ohjelmistoinsinööri Capers Jones kutsuu ilmiötä urbaaniksi legendaksi. Jones väittää, että kyseinen laskutapa ei ota huomioon virheen hinnan ja tuotteen laadun välistä suhdetta, joka muuttuu jokaisen virheen korjauksen myötä. Hän sanoo myös, että metriikka sivuuttaa testauksen aiheuttamia kiinteitä kustannuksia, eikä näin ollen välttämättä kuvasta todellisia taloudellisia faktoja. (Jones 2012, 2, 21.)



Kuvio 1. Virheiden korjauksen hinnan esitetään usein kasvavan eksponentiaalisesti (Kasurinen 2013, 18).

Yhteiskunnastamme on tullut viime vuosikymmenien aikana entistä riippuvaisempi teknologiasta ja ohjelmistoista. Yritysten järjestelmien ja teollisuuden laitteiden lisäksi yhä useammat arkipäiväiset välineet, kuten ajoneuvot ja kodin valaisimet, käyttävät jotain ohjelmistoa ja ovat jatkuvasti kytköksissä verkkoon. (Herrin 2019.) 5G-teknologian kehittyessä verkkoon yhdistetyt laitteet tulevat yleistymään entuudestaan. Ericssonin vuonna 2019 julkaiseman raportin mukaan internetiin yhdistettyjen laitteiden määrä tulee kasvamaan 10,8 miljardista laitteesta 24,9 miljardiin laitteeseen vuoteen 2025 mennessä (Ericsson 2019). Kuinka voimme siis varmistua siitä, että kaikki nämä laitteet, joita käytämme päivittäin, toimivat luotettavasti?

2.2 Miten ohjelmistoja testataan?

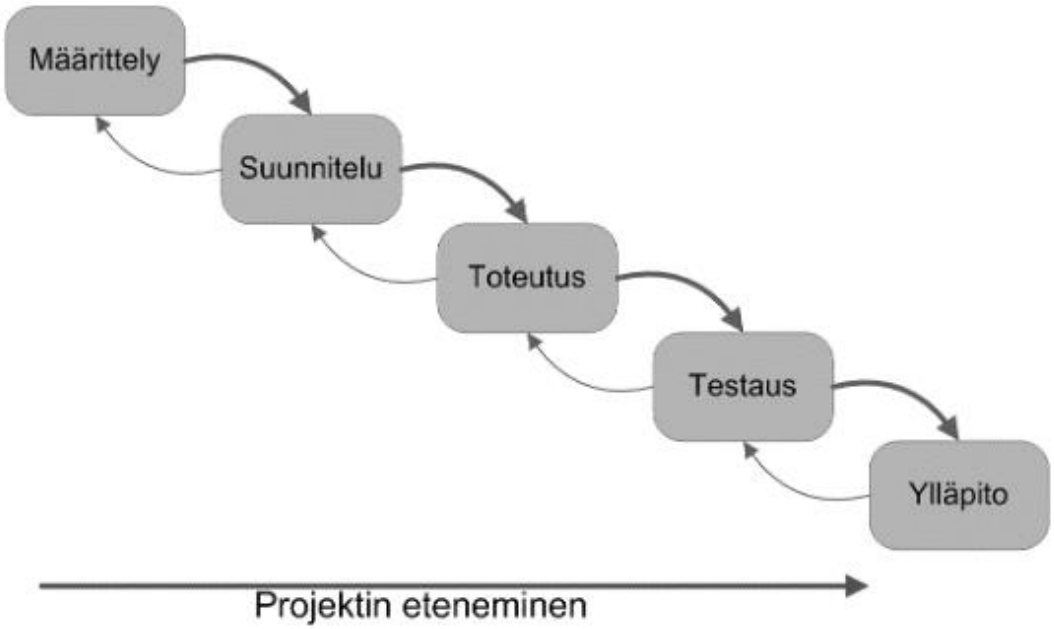
Ohjelmistotestaukseen liittyy paljon eri vaiheita, kuten testaussuunnitelman tekeminen ja testien toteuttaminen. Testaajan työtehtävät voivat vaihdella hyvinkin paljon eri ohjelmistotalojen välillä riippuen siitä, millaista ohjelmistoa ollaan testaamassa. (Kasurinen 2013, 10.) Esimerkiksi lentokoneen ohjausjärjestelmän ja kännykkäpelin testaaminen poikkeaa prosesseina hyvin paljon toisistaan. Ensimmäisessä halutaan varmistua sataprosenttisesti ohjelmiston toiminnasta kaikki

vikatilanteet huomioiden, koska satojen ihmisten henki voi riippua siitä (Sampson 2018). Jälkimmäisessä taas tärkeintä on, että esimerkiksi kaikki graafiset elementit toimivat oikein ja peliä on hauska pelata (Kasurinen 2013, 10).

Menestyksekkään testausprosessin takana on muiden onnistuneiden projektien tavoin huolellinen suunnittelu. Suunnittelulla voidaan määrittää testaukselle saavutettavissa olevat tavoitteet. (Everett & McLeod 2007, 66.) Kun projektin tavoitteet on määritetty, voidaan testausprosessin aikaa ja resursseja arvioida paremmin. Selkeät tavoitteet helpottavat myös tuotteen laadun varmistamisessa ja sen mittaamisessa. (Tuovinen 2013, 17–20.)

2.2.1 Vesiputousmalli

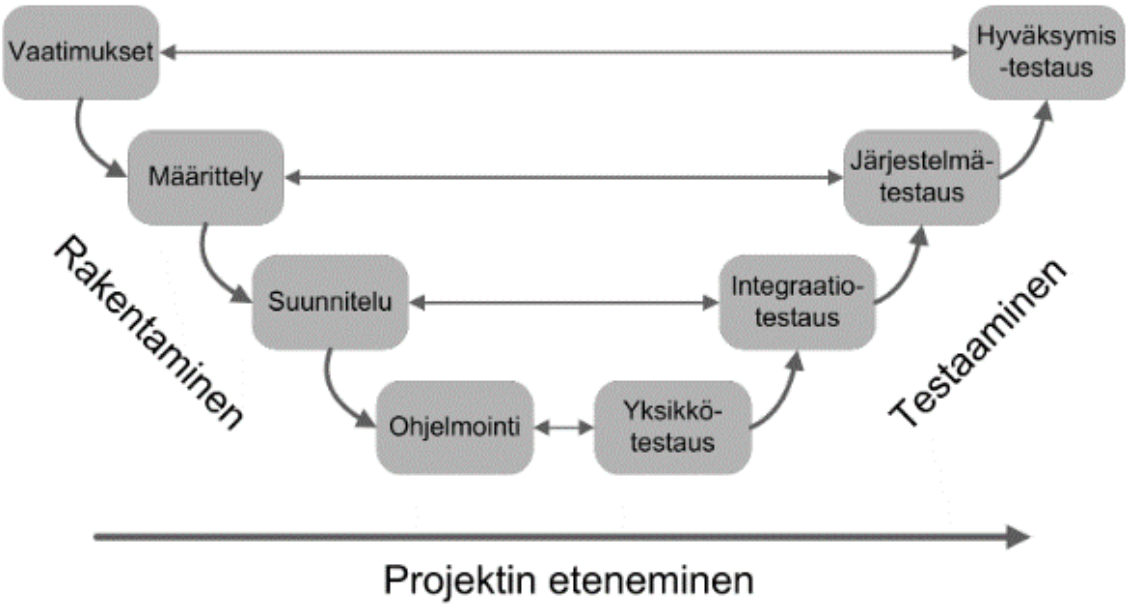
Ohjelmistokehitystä kuvataan usein kolmella eri mallilla, jotka ovat vesiputousmalli, v-malli ja ketterät menetelmät. Vesiputousmalli on perinteinen ohjelmistokehityksen menetelmä, jossa kehityksen eri vaiheet seuraavat toisiaan järjestyksessä (kuvio 2). Vesiputousmallissa testaus on vain yksi työvaihe projektin edetessä. Malli on vanhanaikainen, eikä sen käyttö ole suositeltavaa, sillä testausvaiheeseen päästessä lähes kaikki projektin suunnittelu- ja kehitystyö on jo tehty. Vesiputousmalli toimii hyvin, jos tuotteen vaatimukset voidaan määrittää absoluuttisen tarkasti jo suunnitteluvaiheessa. Näin ei kuitenkaan yleensä ole, joten parempi ratkaisu on käyttää v-mallia tai ketteriä menetelmiä. (Kasurinen 2013, 13–14.)



Kuvio 2. Vesiputousmallissa testaus on vain yksi työvaihe, joka suoritetaan toteutuksen jälkeen (Kasurinen 2013, 13).

2.2.2 V-malli

V-mallissa (kuvio 3) projekti etenee samoin kuin vesiputousmallissakin, mutta testaus ei ole pelkästään yksi vesiputouksen työvaiheista, vaan sen rinnalla samanaikaisesti tapahtuva prosessi. V-mallissa projektin jokaiselle rakennusvaiheelle on määritetty oma testauskategoria. (Kasurinen 2013, 14.) Seuraavaan vaiheeseen siirrytään vasta, kun edellinen vaihe on valmis. Tämä tarkoittaa sitä, että jokaiselle kehitysvaiheelle on olemassa vastaava testausvaihe. V-mallia suositellaankin käytettäväksi vesiputousmallin sijaan, sillä etenkin suurissa projekteissa määritykset ovat niin monimutkaisia, että tärkeitä yksityiskohtia voi jäädä määrittelyvaiheessa huomaamatta. Tämä voi johtaa pahimmassa tapauksessa jopa kokonaan väärän tuotteen toimittamiseen. (Guru99 2020a.)



Kuvio 3. V-mallissa kehitys ja testaus tapahtuvat rinnakkain (Kasurinen 2013, 14).

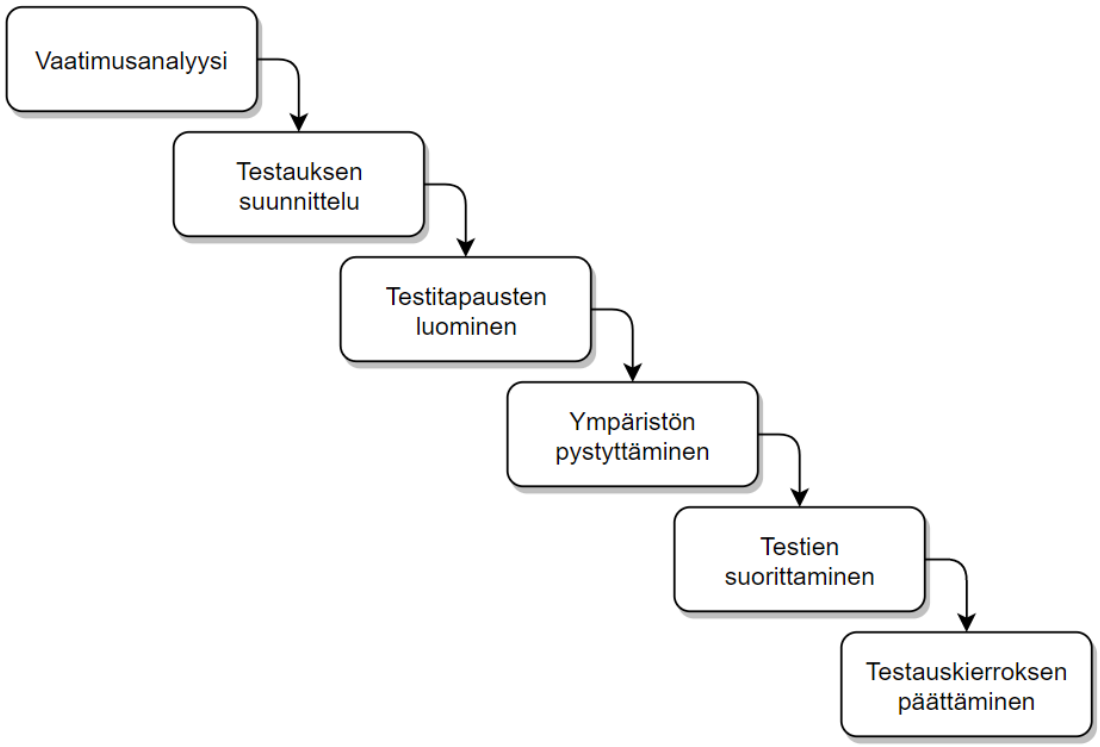
2.2.3 Ketterät projektit

Ketterissä ohjelmistokehitysprojekteissa testaaminen on jatkuva prosessi vaiheittaisen sijaan. Testaus alkaa projektin kanssa samaan aikaan, ja se kulkee kehityksen kanssa käsi kädessä koko projektin ajan. Ketterät testausmenetelmät sopivat hyvin pienille projekteille, joissa määritelmät voivat muuttua ja kehittyä projektin edetessä. (Reqtest 2018).

Ketterissä projekteissa testaus suoritetaan lyhyissä iteraatioissa, jotka ovat yleensä 1-4 viikkoa. Testausta ei varsinaisesti suunnitella etukäteen, vaan jokaisessa iteraatiossa laaditaan uusi testaussuunnitelma sen iteraation aikana toteutettaville tuotteille. (Guru99 2020b). Jatkuvan testauksen etuna on myös siitä saatavat jatkuvat tulokset, jotka edistävät testaus- ja kehitysmenetelmiä tulevissa iteraatioissa. Ketterä testaus säästää myös aikaa ja rahaa sekä vähentää dokumentaatiota. Se on myös huomattavasti vaiheittaista testausta joustavampi ja mukautuvampi prosessi. (Reqtest 2018).

2.3 Ohjelmistotestauksen elinkaari

Ohjelmistotestauksen elinkaarella (kuvio 4) tarkoitetaan testausprosessia, jossa on tietyssä järjestyksessä suoritettavat määritetyt vaiheet. Elinkaaren avulla voidaan varmistua tuotteen laatumääritysten saavuttamisesta. Ohjelmistotestauksen elinkaaren jokaisella vaiheella on oma tavoite sekä tuote, kuten dokumentti vaiheen tuloksista, jonka vaihe päätyttyään tuottaa. Elinkaaren vaiheet voivat vaihdella hieman yrityksittäin, mutta prosessin runko pysyy samana. (Software Testing Help 2019a.)



Kuvio 4. Ohjelmistotestauksen elinkaari eli Software Testing Life Cycle, STLC (Deb 2019 mukaillen).

2.3.1 Vaatimusanalyysi

Ohjelmistotestauksen elinkaari saa alkunsa vaatimusanalyysin tekemisellä. Analyysin tavoitteena on selvittää ohjelmiston vaatimukset siten, että ohjelmisto on mahdollista toteuttaa niiden mukaisesti. Vaatimusanalyysin lopputuloksena on käytettävän kehitysmallin mukaan joko täydellinen lista tuotteen vaatimuksista ja

niiden vaikutuksista tai lista suunniteltavaan iteraatiokierrokseen vaikuttavista vaatimuksista. (Taina 2013, 1–2.)

Vaatimukset jaetaan usein selkeyden vuoksi kolmeen tasoon: käyttäjän vaatimuksiin, järjestelmän vaatimuksiin sekä ohjelmiston määrittelykuvauksiin. Tasojen lisäksi vaatimukset voi tyypittää kahteen kategoriaan, jotka ovat toiminnalliset (functional) vaatimukset sekä ei-toiminnalliset (non-functional) vaatimukset. Vaatimukset ovat yleensä eritasoisia, sillä ne voivat kuvata testitapauksia hyvin korkealla tasolla tai todella yksityiskohtaisesti. Dokumentoitavien vaatimusten tulee olla virheettömiä ja realistisia, eivätkä ne saa olla ristiriidassa keskenään. (Taina 2013, 2–3.)

2.3.2 Testauksen suunnittelu

Seuraava vaihe testauksen elinkaaressa on testauksen suunnittelu, jonka voidaan sanoa olevan prosessin tärkein vaihe. Vaiheesta käytetään joskus myös nimeä testausstrategia. Tämän vaiheen aikana määritetään koko prosessin metriikat, kuten työaika- ja hinta-arviot. Tässä vaiheessa valitaan myös käytettävät testaustasot. (Deb 2019.) Testaustasoja käsitellään tarkemmin tämän työn luvussa 2.4 Testaustasot.

Lisäksi testien suunnitteluvaiheeseen kuuluu käytettävien työkalujen ja -ympäristöjen valitseminen. Suunnittelun lopputuloksena on testaussuunnitelma-dokumentti sekä dokumentoitu arvio koko testausprosessin työmäärästä, kustannuksista sekä henkilöresursseista. (Guru99 2020c.)

2.3.3 Testitapausten luominen

Kun testaussuunnitelma on määritetty, voidaan siirtyä kolmanteen vaiheeseen, joka on testitapausten luominen. Tässä vaiheessa testaustiimi luo yksityiskohtaiset testitapaukset suunnitelman perusteella sekä valmistelee tarvittaessa

testausdatan. Hyvän testitapauksen piirteitä ovat yksinkertaisuus ja läpinäkyvyys. Hyvä testi on luotu loppukäyttäjää ajatellen, eivätkä testit toista itseään.

On myös tärkeää muistaa, että testin tulisi kattaa ohjelmistolle asetetut vaatimukset sataprosenttisesti. Testitapausten ja testausdatan valmistuttua arvioidaan ne joko tiimin kesken tai niitä arvioi laatutiimin johtaja. (Deb 2019.) Arvioinnin jälkeen testejä tai testausdataa voidaan tarvittaessa korjata ja tarkentaa (Guru99 2020c).

2.3.4 Ympäristön pystyttäminen

Tässä vaiheessa prosessia luodaan tarvittava ympäristö testien suorittamista varten. Yleensä ympäristön pystyttäminen tarkoittaa käytettävän laitteiston sekä ohjelmistojen valmistelua testejä varten. Vaiheen tarkoitus on ymmärtää tuotteen arkkitehtuuri ja luoda ympäristö, jossa laitteisto, ohjelmisto ja verkkoyhteys keskustelevat keskenään.

Testiympäristö voi olla esimerkiksi testidataa tietokantaan tallentava käyttöliittymä, jota käytetään tietokoneen selaimella. (Guru99 2020d.) Yleinen lähestymistapa on käyttää testidatana kopiota yrityksen tuotannossa olevasta datasta (Deb 2019).

2.3.5 Testien suorittaminen

Elinkaaren viides vaihe on kolmannessa vaiheessa laadittujen testitapausten suorittaminen. Vaiheen tarkoitus on suorittaa testit testausstrategiassa määritetyllä tavalla ja analysoida tuloksia. Tulosten analysointi tarkoittaa käytännössä sitä, että testien odotettuja tuloksia verrataan todellisiin tuloksiin. Testauksen aikana löydetyt virheet korjataan, ja testit suoritetaan uudelleen. (Guru99 2020c.)

Testitapaukset luokitellaan suorituksen perusteella. Luokkia voivat olla esimerkiksi testi on onnistunut, testi on epäonnistunut tai testiä ei ole suoritettu. Testien suorittamisesta laaditaan testitapausten suoritusraportti. Lisäksi

epäonnistuneiden tai odottamattomia tuloksia antaneiden testien avulla löydetyistä virheistä laaditaan virheraportti. (Deb 2019.)

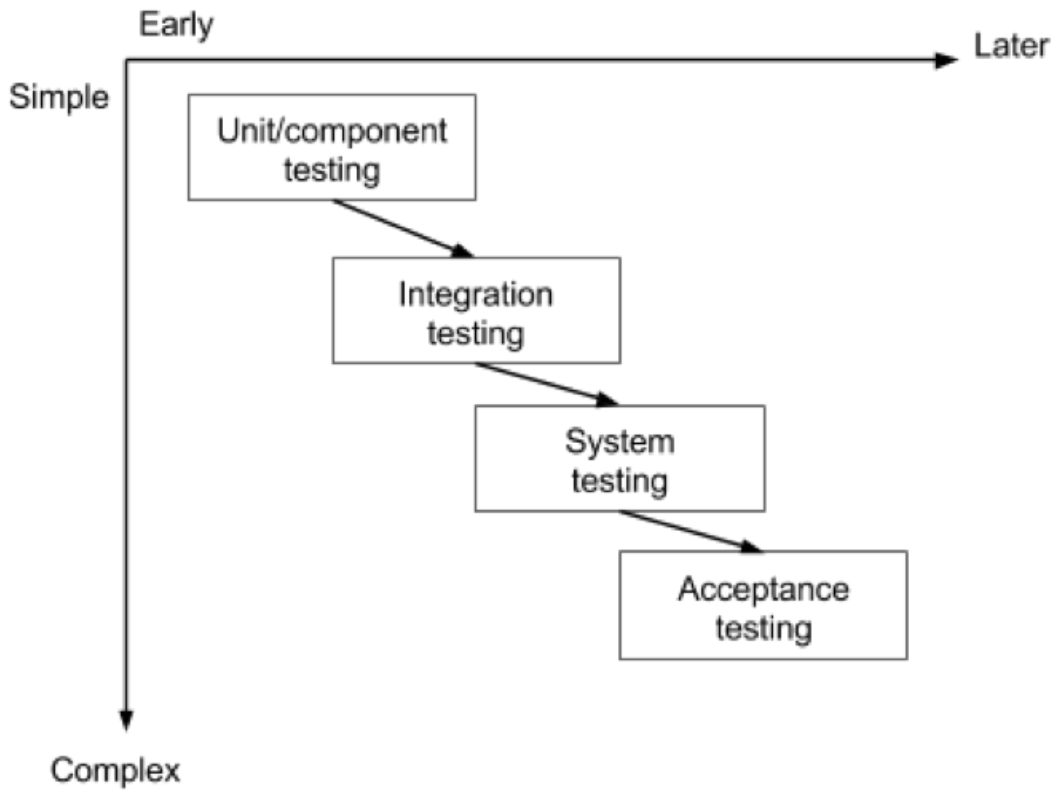
2.3.6 Testauskierroksen päättäminen

Testausprosessin viimeinen vaihe on testauskierroksen päättäminen. Päättämisvaiheen tarkoitus on analysoida koko prosessin tuloksia ja tehdä loppuraportti.

Elinkaaren viimeisessä vaiheessa testaustiimi kokoontuu ja arvioi sitä, kuinka testaussykli on onnistunut ajankäytön, testikattavuuden, kulujen, ohjelmiston, liiketoiminnan ja laadun perusteella. Raportin tarkoitus on myös reflektoida prosessin onnistumisia ja epäonnistumisia ja helpottaa tiimin valmistautumista seuraavaa testaussykliä varten. (Guru99 2020c.)

2.4 Testaustasot

Suoritettavat testit jaetaan usein neljään tasoon (kuvio 5), jotka ovat yksikkötestaus, integraatiotestaus, järjestelmätestaus sekä hyväksymistestaus. Testien suorittamisen tulisi tapahtua loogisessa järjestyksessä yksikkötestauksen ollessa ensimmäinen suoritettava taso ja hyväksymistestauksen viimeinen. Testien suorittamisvaihe ja monimutkaisuus ovat kytköksissä toisiinsa. Kehityksen alkuvaiheessa toteutettavat yksikkötestit ovat yksinkertaisia, kun taas loppuvaiheessa suoritettava hyväksymistestaus on haastavaa ja aikaa vievää, aiheuttaen näin ollen myös enemmän kustannuksia. (Eriksson 2014a.)



Kuvio 5. Ohjelmistotestauksen neljä tasoa (Eriksson 2014a).

2.4.1 Yksikkötestaus

Yksikkötestaus (unit testing), josta käytetään joskus myös nimitystä komponenttitestaus, on kaikista testaustoimenpiteistä yleisin. Yksikkötestauksen tavoite on testata järjestelmän jokaista osaa yksittäisenä komponenttina. Komponenttia testataan eristettynä järjestelmän muusta toiminnasta, jotta voidaan varmistua, että komponentti täyttää sille asetetut vaatimukset ja se toimii halutulla tavalla. (Eriksson 2014a.)

Yksikkötestauksen suorittaa yleensä itse ohjelmoija toteutuksen yhteydessä. Ohjelmointivaiheessa huomattu vika on helppo korjata ennen kuin komponentti ehtii edes käydä osana suurempaa järjestelmää. (Kasurinen 2013, 51–52.)

Yksikkötestauksen haastavuutta lisää kuitenkin se, että yksittäinen komponentti ei yleensä pysty tekemään mitään itsenäistä. Useimmiten komponentit keskustelevat keskenään, kuten käyttöliittymäkomponentti voi toteuttaa http-pyynnön

(http-protokollaa käyttävä verkkopyyntö), johon järjestelmä reagoi tallentamalla haetun tiedon tietokantaan. Tämmöisissä tapauksissa yksikkötestin kirjoittamista voidaan helpottaa rakentamalla testikomponentteja tai testitynkiä (mocks, stubs). Testikomponenttien tarkoitus on simuloida komponenttien välistä toimintaa järjestelmässä. Ohjelmoija voi hallita sitä, mitä tietoa testikomponentit lähettävät, jolloin itse testattava komponentti tulee testattua varmasti halutulla tavalla. (Kasurinen 2013, 52.)

2.4.2 Integraatiotestaus

Kun yksikkötestaus on suoritettu ja on todettu, että yksittäiset komponentit toimivat kuten pitää, voidaan siirtyä seuraavalle tasolle, eli integraatiotestaukseen (integration testing). Integraatiotestauksen tarkoitus on aloittaa toteutettujen komponenttien yhteen sovittaminen ja testata niiden toimintaa yhdessä. Vaiheen tavoite on saada koottua komponentit yhdeksi toimivaksi kokonaisuudeksi. Myös integraatiotestauksessa voidaan joutua käyttämään tynkiä, mikäli joku testattavista osista sisältää toteuttamattomia toimintoja. Integraatiotestausta on esimerkiksi kahden samaa tietokantaa käyttävän komponentin yhteistoiminnan testaaminen. (Kasurinen 2013, 54.)

Integraatiotestauksessa integrointijärjestys voidaan toteuttaa kolmella eri tavalla, jotka ovat alhaalta ylöspäin testaus (bottom up testing), ylhäältä alaspäin testaus (top down testing) sekä voileipätestaus (sandwich testing) (Kasurinen 2013, 55).

Alhaalta ylöspäin -metodissa integrointi aloitetaan kaikista matalimman tason moduuleista, kuten tietokannasta ja verkkoyhteydestä siirtyen aiempia moduuleita hyödyntäen järjestelmäarkkitehtuurissa ylöspäin. Metodin etuna on, että matalan tason virheitä löydetään helposti. Ylhäältä alaspäin -menetelmä toimii samoin, mutta päinvastaisessa järjestyksessä. Ylhäältä alaspäin -menetelmä antaa testattavasta järjestelmästä paremman yleiskuvan, mikä helpottaa esimerkiksi puuttuvien ominaisuuksien löytämisessä. Voileipätestaus hyödyntää molempia edellä mainittuja integrointitapoja siten, että integraatiota lähestytään

molemmista suunnista samanaikaisesti. Voileipämenetelmän tuoma etu on se, että tynkien käyttöä tarvitaan usein vähemmän. (Kasurinen 2013, 55.)

2.4.3 Järjestelmätestaus

Testauksen kolmas taso on järjestelmätestaus (system testing). Tässä vaiheessa järjestelmän kaikkia osia testataan yhtenä kokonaisuutena, kun ne ovat edellisen vaiheen jäljiltä integroitu onnistuneesti. Järjestelmätestauksen tarkoitus on varmistaa, että kokonaisen järjestelmän toiminnot vastaavat vaatimuksia. (Eriksson 2016.)

Järjestelmätestauksella ei tarkoiteta varsinaisesti mitään tiettyä testaustapaa, vaan se on yleisnimitys testaukselle, jota tehdään kokonaiselle järjestelmälle, jossa ei ole enää mukana tynkiä tai muita testikomponentteja. Järjestelmätestauksen aikana toteutettavat testitapaukset voivat vaihdella projekteittain, mutta yleensä vaiheen aikana tehdään musta laatikko ja lasilaatikko -testausta, käyttäjätestausta, kuormitustestausta tai mitä tahansa toiminnallista kokonaisuutta tarkastelevaa menetelmää. (Kasurinen 2013, 56–57.)

2.4.4 Hyväksymistestaus

Hyväksymistestaus (acceptance testing) on viimeinen testaustaso. Tämän testaustason tarkoitus on varmistua siitä, että toteutettu ohjelmisto vastaa määritettyjä vaatimuksia on valmis julkaistavaksi tuotantoympäristöön.

Hyväksymistestausta voidaan tehdä yrityksen sisäisesti, kuitenkin sellaisten henkilöiden toimesta, jotka eivät ole olleet mukana kehittämässä järjestelmää. Tällöin kyseessä on järjestelmän alpha-testaus. Vaihtoehtoisesti hyväksymistestausta voidaan tehdä yhdessä loppukäyttäjän kanssa, jolloin kyseessä on betatestaus. Kumpikin tapa tuo testaus- ja kehitystiimille arvokasta tietoa järjestelmän toiminnasta. Jos järjestelmästä löytyy hyväksymistestauksen aikana vikoja, ne korjataan ja korjaukset todennetaan ennen julkaisua. (Eriksson 2014b.)

Hyväksymistestaus on valmis, kun asiakas on hyväksynyt tuotteen valmistumisen. Tämän jälkeen tuote siirtyy juridisesti asiakkaan omistukseen. (Kasurinen 2013, 57.)

2.4.5 Regressiotestaus

Edellisistä poiketen regressiotestaus ei ole varsinaisesti erillinen testaustaso, vaan testauksessa käytettävä yleisnimitys. Regressiotestauksella tarkoitetaan ohjelmistokehityksessä jonkin osa-alueen uudelleentestaamista (Kasurinen 2013, 68).

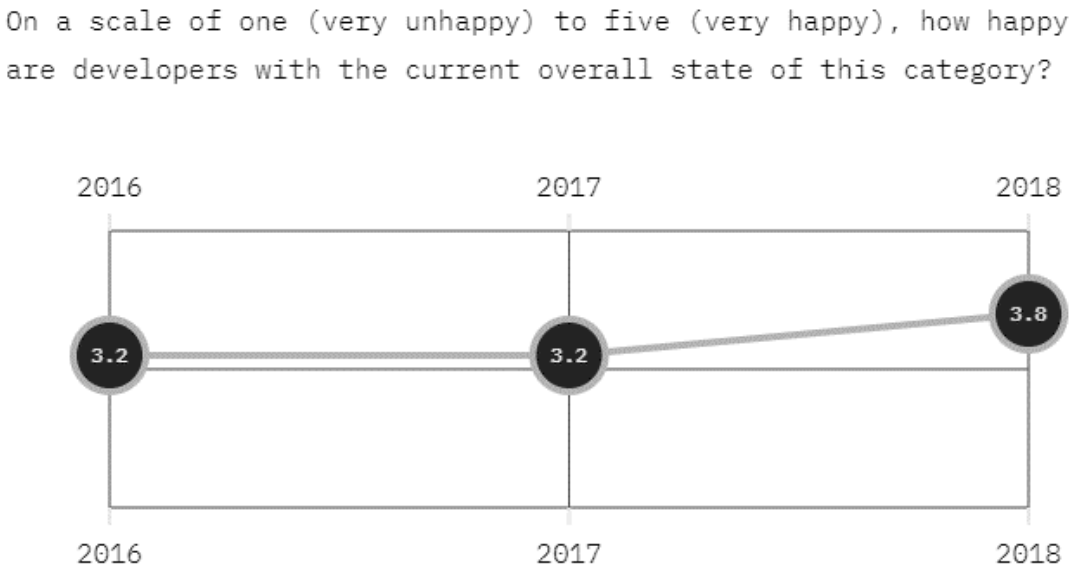
Kun ohjelmistoon lisätään uusi osio tai sitä muutetaan, saattaa aikaisemmin virheettömästi toiminut funktio yllättäen aiheuttaakin virheen. Regressiotestauksen avulla pyritään löytämään ja korjaamaan tällaiset muutosten aiheuttamat virheet. Termillä tarkoitetaankin yleensä tiettyjen testien uudelleensuorittamista, jotta voidaan varmistua siitä, että uudet muutokset koodissa eivät aiheuta yllättäviä vikoja tai muuta ei-haluttua toimintaa ohjelmistossa. (Pressman 2010, 462.)

3 Testaus Javascript -kehityksessä

Verkkosovellukset ovat kehittyneet kuluneen vuosikymmenen aikana valtavasti, ja yhä useammat yritysten ja teollisuuden järjestelmät toimivat selainpohjaisina verkkosovelluksina, kuten Software as Service (SaaS) -palveluina (Politis 2017). Verkkosovellusten kehittämiseen käytetään lähes aina Javascript-ohjelmointikieltä, joka mahdollistaa dynaamisten verkkosivujen toteuttamisen (MDN Web Docs, 2019). Javascript-kehityksessä käytetään yleensä apuna kirjastoiksi kutsuttuja moduuleita, joiden tarkoitus on nopeuttaa kehitystä käyttämällä valmiiksi kirjoitettua koodia. (Khan Academy 2019). Suosittuja kirjastoja ovat esimerkiksi tässä opinnäytetyössä käsiteltävät käyttöliittymien toteutukseen tarkoitettu React.js sekä palvelinympäristö Node.js (Stack Overflow 2019).

Vaikka web-järjestelmien testaaminen on ollut mukana kuvioissa yhtä kauan kuin järjestelmätkin, eivät testauskehykset ole pysyneet nopeasti muuttuvien ja modernien web-teknologioiden tahdissa. Viime vuosien aikana Javascript-kirjastojen testaus on kuitenkin ottanut askeleen eteenpäin ja sen on sanottu muuttuneen hitaasta, hankalasta ja tylsästä työstä nopeaksi ja mielenkiintoiseksi. (Zaidman 2019.)

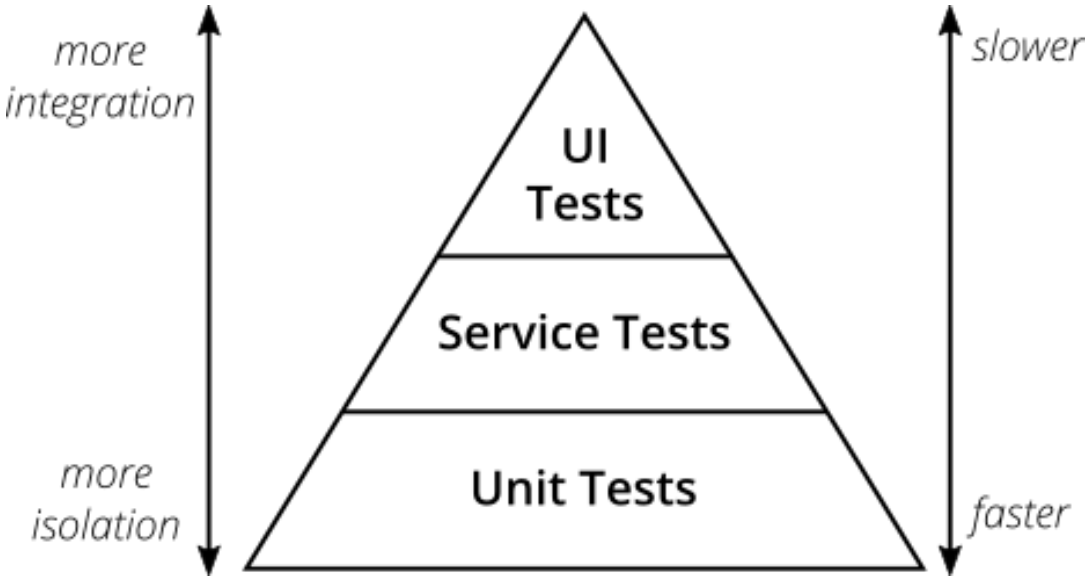
Javascriptin testauksen haastavuus on johtunut osittain siitä, että testauskehyksiä on paljon ja ne ovat vielä jokseenkin hajallaan verrattuna Javascriptin front- ja back-end kirjastoihin, joihin on muutaman kuluneen vuoden aikana asettunut hallitsevat kirjastot (Greif, Benitte & Rambeau 2018a). Testauskehyksissä on kuitenkin havaittavissa vastaavanlainen tilanne, jossa tietyt kehykset nousevat ylitse muiden. Etenkin Facebookin Jest, joka mahdollistaa sekä käyttöliittymän että palvelimen testauksen, on kasvattanut suosiotaan räjähdysmäisesti. (Greif ym. 2018b.) Vuonna 2018 julkaistun The State of Javascript -kyselyn mukaan kehittäjien onnellisuus testauskategoriassa nousikin vuodesta 2017 18,75 prosenttia (kuvio 6). (Greif ym. 2018c.)



Kuvio 6. Kehittäjien onnellisuus testauskategoriassa (Greif ym. 2018c).

Verkkosovellusten testauksen voi jakaa automatisoitavien testien perusteella kolmeen tasoon. Nämä tasot ovat yksikkötestit, integraatiotestit sekä käyttöliittymä-

tai funktionaaliset eli end-to-end (E2E) testit. (Elliot 2016). Näiden kolmen tason kokonaisuutta kutsutaan usein myös nimellä testauspyramidi (kuvio 7). Testauspyramidi on Mike Cohnin kehittämä malli, joka kuvaa eri tasoissa tarvittavien testien määrää sekä niiden monimutkaisuutta kussakin tasossa. Mallin mukaan yksikkötestien toteutus on nopeaa ja niitä pitäisi olla paljon, kun taas käyttöliittymän testaus on hidasta tarvittavien integraatioiden takia. Mitä korkeammalle pyramidissa kavutaan, sitä vähemmän testejä tulisi toteuttaa. Mallia on myös moitittu sen liiallisen yksinkertaisuuden vuoksi, mutta sen on sanottu olevan siitä huolimatta hyvä nyrkkisääntö testauksen toteutusta varten. (Vocke 2018.)



Kuvio 7. Testauspyramidi (Vocke 2018).

Yksikkö- ja integraatiotestaus toteutetaan myös verkkosovelluksissa kuten ne on kuvattu kappaleessa 2.4 Testaustasot. Käyttöliittymä- ja end-to-end -testauksen sanotaan joskus olevan sama asia, mikä ei kuitenkaan aina pidä paikkaansa. Käyttöliittymän testaus voi olla yksikkötestausta, mikäli käyttöliittymän osia on mahdollista eristää muusta sovelluksesta testattavaksi. End-to-end testaus sen sijaan testaa sovelluksen käyttöliittymää, joka voi olla integroitu muihin palveluihin. Verkkosovelluksissa end-to-end testaus tarkoittaa käytännössä sitä, että testaustyökalu pyörittää sovellusta selaimessa ja toteuttaa ennalta määriteltyjä toimintoja, kuten elementtien klikkailua, tiedonsyöttöä tai käyttöliittymän tilan tarkastusta. End-to-end testit voivat olla kuitenkin ongelmallisia. Koko järjestelmän pystyttämisen toimiakseen vaativien testien kehittäminen ja suorittaminen on

hidasta, joten testaus vie paljon aikaa ja on sen vuoksi myös kallista. Testien ylläpito voi olla myös työlästä, ja integraatioiden vuoksi saattaa olla vaikea määrittää kuka niiden toteutuksesta on vastuussa. End-to-end testejä tulisikin käyttää testauspyramidin mukaisesti vain vähän ja testien pitäisi kattaa vain verkkosovelluksen tärkeimmät toiminnot. Usein pyramidin alemmat tasot kattavat jo monet tärkeät ääritapaukset ja integraatiot, eikä näitä testejä ole yleensä syytä toistaa käyttöliittymän kanssa uudestaan. (Vocke 2018).

4 Projektin esittely

4.1 Työaikapankki

Tässä opinnäytetyössä testattava sovellus on toimeksiantajan jo olemassa olevaan työajanhallintajärjestelmään lisäpalveluna toteutettava työaikapankki. Työaikapankki on uuden työaikalain (872/2019) määrittämä palvelu, joka mahdollistaa työajan säästämisen ja sen yhdistämisen muuhun ansaittuun vapaa-aikaan. Työaikalaissa (872/2019) työaikapankki on kuvattu seuraavasti:

> Työaikapankilla tarkoitetaan tässä laissa työ- ja vapaa-ajan yhteensovitusjärjestelmää, jolla työaikaa, ansaittuja vapaita tai vapaa-ajaksi muutettuja rahamääräisiä etuuksia voidaan säästää ja yhdistää toisiinsa. Työnantaja ja luottamusmies tai jos sellaista ei ole valittu, luottamusvaltuutettu tai muu työntekijöiden edustaja taikka henkilöstö tai henkilöstöryhmä yhdessä saavat sopia kirjallisesti työaikapankin käyttöönotosta. Edustajan tekemä työaikapankkia koskeva sopimus sitoo niitä työntekijöitä, joita edustajan katsotaan edustavan. (14. §.)

Työaikapankin toteutus on toimeksiantajan tapauksessa selaimessa käytettävä verkkosovellus. Sen palvelinpuoli on rakennettu Node.js:llä ja käyttöliittymä React.js:llä. Työaikapankin keskeisin ominaisuus, jota tässä työssä testattiin, on tuntien lisääminen pankkiin. Seuraavissa kappaleissa perehdytään hieman tarkemmin sovelluksen taustalla oleviin teknologioihin sekä opinnäytetyön käytännön osiossa käytettyihin työkaluihin.

4.2 Node.js

Node.js on avoimen lähdekoodin asynkroninen ja tapahtumavetoinen Javascript-ajoympäristö (Cuomo 2013). Asynkronisuudella tarkoitetaan ohjelmoinnissa sitä, että ohjelma ei suorita tapahtumia järjestyksessä rivi kerrallaan, vaan monta asiaa voi tapahtua yhtä aikaa. Synkroninen ohjelmakoodi joutuu odottamaan esimerkiksi funktion päättymistä, ennen kuin ohjelma voi jatkua. Asynkroninen ohjelmointi sen sijaan mahdollistaa prosessorin suorittaa muuta koodia sillä aikaa, kun toisen toiminnon suoritus on käynnissä. (MDN Web Docs 2020.) Node.js on suunniteltu käytettäväksi skaalautuvien verkkosovellusten, etenkin web-palvelimien, toteuttamiseen. (Cuomo 2013).

Perinteisesti web-palvelimet toimivat siten, että jokainen pyyntö palvelimelle muodostaa uuden säikeen tai prosessin pyynnön käsittelyä varten. Jokainen säie tai prosessi kuluttaa palvelimen resursseja, kuten muistia ja laskutehoa. Lisäksi pyynnön suorittamisen aikana palvelin ei tee mitään muuta työtä, vaan odottaa, että pyyntö on valmis ennen ohjelman jatkamista. Vaikka säikeiden käyttö on prosesseihin verrattuna suhteellisen tehokasta, niiden käyttö suuressa mittakaavassa voi aiheuttaa viivettä. Tästä syystä monisäikeinen palvelin voi rajoittaa sovelluksen skaalautuvuutta. (Heller 2017.) Node.js toimiikin poikkeuksellisesti Javascriptin tapaan yksisäikeisesti (NodeJS 2020a). Nodessa jokainen pyyntö lisätään tapahtumaketjuun (event loop), joka mahdollistaa yksisäikeisyydestä huolimatta useiden pyyntöjen yhtäaikaisen suorittamisen siirtämällä operaatiot monisäikeistä teknologiaa hyödyntävien ytimien (kernel) suoritettaviksi (NodeJS 2020b). Yksisäikeinen ja tapahtumavetoinen teknologia mahdollistaa kymmenien tuhansien yhtäaikaisten pyyntöjen säilyttämisen luomatta uusia säikeitä, mikä tekee Nodesta erittäin skaalautuvan ratkaisun web-palvelimen toteutusta varten (Capan 2013).

Node.js sai alkunsa, kun yhdysvaltalainen ohjelmistoinsinööri Ryan Dahl sai ajatuksen luoda asynkronisesti ja jaksottomasti (non-blocking) toimiva palvelin (Shashidhara 2017). Dahl yritti alun perin toteuttaa palvelinta C-, Lua- sekä Haskell-ohjelmointikielillä, mutta projektit epäonnistuivat hänen huomatessaan, etteivät kielet soveltuneet hänen tarpeisiinsa. Samoihin aikoihin Google julkaisi

Google Chrome -selaimen yhdessä Chromium Projectin V8 Javascript-moottorin kanssa. Dahl tajusi tällöin yhtäkkiä, että Javascript olisi täydellinen valinta hänen suunnittelemansa palvelimen kehittämiseen. (McCarthy 2011.) Dahl esitteli Node.js:n ensimmäisen kerran vuonna 2009 Berliinissä järjestetyssä JSConf-tapahtumassa (JSConf 2013).

Vuonna 2010 Dahl ilmoitti, että Node.js on siirtynyt pilvipalveluyritys Joyentin omistukseen. Hän kuitenkin sanoi jatkavansa Noden kehittämistä yrityksen palveluksessa. (Dahl 2010.) Kolme vuotta myöhemmin Dahl kirjoitti siirtyvänsä syrjään Noden kehityksestä voidakseen keskittyä paremmin tutkimusprojekteihin. Node.js:n pääkehittäjän titteli siirtyi Dahlin jälkeen Isaac Schlueterille. (Dahl 2013.)

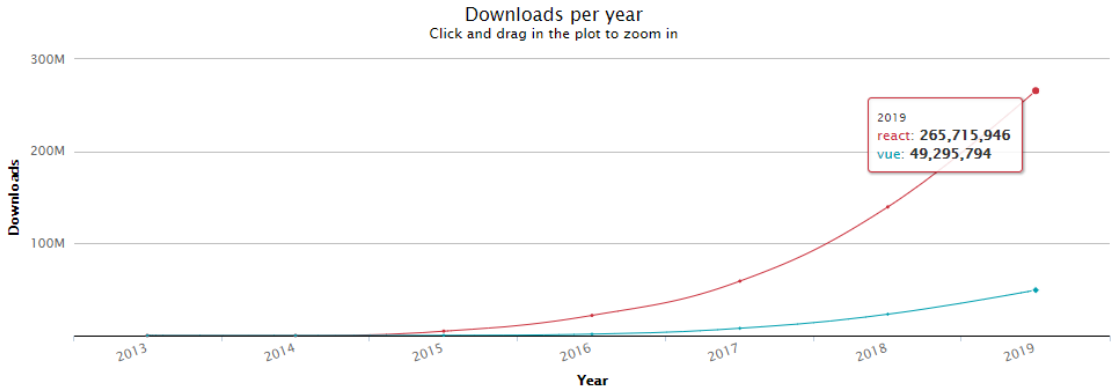
Vuoden 2018 JSConf-tapahtumassa Ryan Dahl palasi lavalle pitämään puheen ominaisuuksista, joiden kehittämistä Nodeen hän katuu. Samassa puheessa Dahl ilmoitti, että hänellä on kehitteillä uusi Javascript-ajoympäristö, Deno, jossa hän pyrkii korjaamaan Noden kanssa tekemiään virheitä. (JSConf 2018.) Denon uusia ominaisuuksia ovat muun muassa suora Typescript-tuki, sekä parannettu turvallisuus. Deno ei myöskään tarvitse erillistä paketinhallintajärjestelmää, kuten Node.js:n kanssa käytettävä Node Package Manager (npm), vaan se toimii itsessään myös paketinhallintajärjestelmänä. (Deno 2019.)

4.3 React.js

React.js on interaktiivisten käyttöliittymien kehittämiseen tarkoitettu Javascript-kirjasto (Facebook 2019a). Reactin kehitti alun perin yhdysvaltalainen Facebookin insinööritiimissä työskentelevä Jordan Walke vuonna 2011 helpottaakseen verkkosivujen ja -sovellusten käyttöliittymien toteuttamista. Vuonna 2013 Facebook julkaisi Reactin avoimen lähdekoodin kirjastona. (Dash Magazine 2019.)

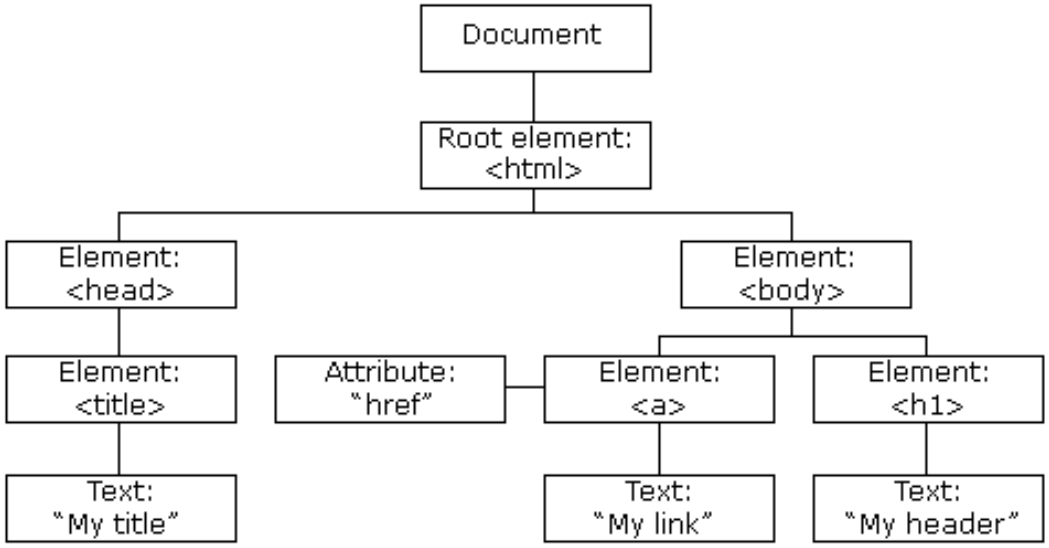
React alkoi saavuttaa suurta suosiota vuoden 2016 aikana, ja sen suosio kasvoikin Stack Overflow:n samana vuonna järjestämän kyselyn mukaan yli 300 % vuodessa (Stack Overflow 2016). Tämän jälkeen Reactin suosio on jatkanut

vauhdikasta kasvuaan. Vuonna 2019 React ylitti 250 miljoonan vuosittaisen latauksen rajan, latausten kokonaismäärän lähennellessä puolta miljoonaa latausta. (Npm-stat 2019.) Reactin pahin haastaja on tällä hetkellä toinen Javascript-käyttöliittymäkirjasto, Vue.js, joka on jo ohittanut Reactin Github-tähtien määrässä. Tätä pidetään kuitenkin yleisesti epäluotettavana mittarina teknologian suosiosta. (Krotoff 2019.) Todellisemman kuvan suosiosta antaa esimerkiksi kirjastojen latausmäärät (kuvio 8), joita tarkastelemalla huomataan, että React on edelleen ylivoimaisesti suosituin Javascript-kirjasto käyttöliittymien kehitykseen (Npm-stat 2019). Reactin suosiosta kertoo myös sitä hyödyntävien yritysten maine. Facebookin ja Instagramin lisäksi Reactiin omissa sovelluksissaan luottavat teknologiajätit kuten Airbnb, Uber, Netflix ja Twitter. (Stackshare 2019).



Kuvio 8. Latausten määrä vuosittain, React vs. Vue (Npm-stat 2019).

Yksi Reactin keskeisimmistä ominaisuuksista on sen hyödyntämä virtuaalinen Domain Object Model (DOM) -konsepti (Facebook 2019b). Perinteinen DOM eli dokumenttioliomalli on W3C-konsortion standardi, jolla kuvataan Hypertext Markup Language (HTML) -verkkosovelluksissa käyttöliittymä puu-tietotyyppinä (kuvio 9). DOM kuvaa kaikki HTML-elementit olioina. Se kuvaa myös elementtien ominaisuudet, metodit ja tapahtumat. Dokumenttioliomalli sallii HTML-sovelluksissa elementteihin pääsyn sekä niiden muokkaamisen Javascriptin avulla. DOM:ia hyödyntämällä Javascriptillä pystyy siis käsittelemään kaikkia HTML-sivun elementtejä ja niiden ominaisuuksia, mikä mahdollistaa dynaamisten verkkosivujen toteuttamisen. (W3Schools 2019.)



Kuvio 9. Esimerkki HTML-dokumenttioliomallista (W3Schools 2019).

Dokumenttioliomallin muokkaus, joka tunnetaan myös manipulaationa, on kuitenkin usein hidasta. Kun elementtiä muokataan, täytyy elementti ja kaikki sen lapsielementit uudelleenrenderöidä sovelluksessa. Mitä enemmän käyttöliittymäkomponentteja sovelluksessa on, sitä hitaampaa uudelleenrenderöinnistä tulee. (Hamedani 2018.)

Reactin käyttämä virtuaalinen DOM on ratkaisu tähän ongelmaan. VDOM on konsepti, jossa dokumenttioliomallista tehdään representaatio eli kevyt kopio. VDOM-oliolla on kaikki samat ominaisuudet, kuin oikealla DOM:illa, mutta sillä ei ole valtuuksia muokata sovelluksen elementtejä. React hyödyntää VDOM:ia komponenttien päivityksessä. Kun React-komponentin tila muuttuu, myös VDOM päivittyy. Päivityksen jälkeen React vertaa päivittynyttä VDOM-oliota päivitystä edeltäneeseen VDOM-olioon. Näitä kahta DOM-representaatioita vertaamalla React tietää tarkalleen mitkä ominaisuudet VDOM-puussa ovat muuttuneet. Kun tämä on tiedossa, React voi päivittää muuttuneet elementit oikealle dokumenttioliomallille, joka päivittää elementit sovelluksessa. Tämän konseptin avulla ainoastaan tärkeät osat dokumenttioliomallista päivitetään, mikä vähentää turhaa uudelleenrenderöintiä huomattavasti. (Codecademy 2019.)

5 Node.js testauskehykset

5.1 Mocha

Mocha on ilmainen avoimen lähdekoodin Javascript-testauskehikko, joka mahdollistaa asynkronisten testien ajamisen selaimessa ja Node.js -projekteissa. Mochan slogan sanoo testauskehikon olevan hauska, yksinkertainen ja joustava. Mochaa käytetään useimmiten yhdessä erillisten todennuskirjastojen (assert library) kanssa, koska siinä itsessään ei ole sisäänrakennettua todennusta. Suosittuja todennuskirjastoja ovat muun muassa Chai.js, Expect.js sekä Should.js. (Mocha 2020a.)

Vuoden 2018 State of Javascript -kyselyn mukaan Mocha oli kaikista eniten käytetty Javascript-testaustyökalu. Mochan suosio selittyy osittain sillä, että se on ollut olemassa jo kauan ja se on kasvattanut ympärilleen laajan ekosysteemin. Pitkään kuvioissa olleena, Mocha on myös monille Javascript-kehittäjille entuudestaan tuttu työkalu. (Greif ym. 2018a.)

Mocha valittiin osaksi opinnäytetyön Node.js -testausosiota, koska työkalu on suosittu ja se sopii loistavasti rajapintojen testaukseen suoran asynkronisen Node.js -tuen vuoksi. Todennuskirjastoksi tähän työhön yhdessä Mochan kanssa käytettäväksi valittiin Chai.js, koska se on Expect.js:n ohella suosituin todennuskirjasto (Npm-stat 2020.) Expectiä sen sijaan käytetään yhdessä Jestin kanssa, joten Chai on hyvä valinta Mochalla toteutettavaan testitapaukseen. Lisäksi, koska Mochan testit suoritetaan komentorivillä, se on helppo integroida myöhemmin CI/CD -putkeen (continuous integration & continuous delivery pipeline). Tämä on testien jatkokehityksen kannalta tärkeää, koska testien suorittaminen putkessa auttaa löytämään virheet helpommin ja parantaa tuotteen laatua (Synopsys 2019).

5.2 Jest

Jest on Facebookin kehittämä ja ylläpitämä Javascript-testauskehikko. Mochan tapaan myös Jest on ilmainen ja sen lähdekoodi on kaikille avoin. (Facebook 2020a.) Jest pyrkii yksinkertaisuuteen, nopeuteen ja turvallisuuteen. Jest on yhteensopiva suosituimpien Javascript-kirjastojen kanssa, ja sillä voi testata muun muassa Node-, Typescript-, Vue- ja React -projekteja. Työkalun sanotaan lisäksi toimivan ilman konfiguraatiota suurimmassa osassa Javascript-projekteja. (Facebook 2020b.)

Jest valikoitui yhdeksi opinnäytetyössä käytettävistä testaustyökaluista, koska se on yksi tämän hetken suosituimmista Javascript-testauskehikoista (Greif ym. 2018c). Jest on myös toimeksiantajan projekteissa valmiiksi käytössä, mutta sillä ei ole toteutettu lainkaan testejä palvelinpuolelle, vaan työkalua on käytetty vain React-komponenttien yksikkötestaukseen. Opinnäytetyö tarjosikin oivan mahdollisuuden myös Jestin palvelinpuolen testaustoimintojen tutkimiselle. Mochan tavoin komentorivityökaluna Jestin lisääminen CI/CD -putkeen on helppoa. Toteutuksessa oli mielenkiinoista nähdä, miten Jest pärjäsi uudempana tulokkaana asemansa jo Javascript-testausmaailmassa vakiinnuttanutta Mochaa vastaan.

5.3 vREST

VREST on automatisoituun represantional state transfer (rest) -rajapintatestaukseen tarkoitettu työkalu. VREST poikkeaa Mochasta ja Jestistä suuresti testien toteutustavassa, sillä vREST-testit toteutetaan ja suoritetaan palvelun käyttöliittymän avulla ohjelmakoodin kirjoittamisen ja komentorivillä suorittamisen sijaan. Palvelu mahdollistaa myös testien nauhoittamisen testattavaa sovellusta käyttämällä, jolloin palvelu muodostaa testit automaattisesti. (vREST 2020a.) Kahdesta muusta tässä opinnäytetyössä käytettävistä Node.js testaustyökaluista poiketen vREST on maksullinen työkalu (vREST 2020b).

VREST valittiin opinnäytetyön kolmanneksi Node.js -testaustyökaluksi, koska se on toimeksiantajan projekteissa valmiiksi käytössä. Lisäksi myös vREST

mahdollistaa testien integroimisen osaksi CI/CD -putkea. (vREST 2020a.) Toteutusosiossa oli mielenkiintoista nähdä, miten käyttöliittymän avulla toteutettavien testien luominen erosi perinteisemmistä testaustyökaluista.

6 Node.js rajapinnan testaus

6.1 Testitapaus

Opinnäytetyön Node.js -testitapauksissa testattiin työaikapankki-sovelluksen rest-rajapintaa. Rest on ohjelmointirajapintojen toteuttamiseen tarkoitettu arkkitehtuurimalli, joka määrittelee, millaisilla operaatioilla tietoa käsitellään palvelimella. Rest-pyyntöjä toteutetaan yhdessä http-protokollan avulla. Pyyntöjä voivat olla esimerkiksi protokollan määrittelemät get, post tai patch. Rest ei sen sijaan määrittele sitä, missä muodossa data tulisi palauttaa. (Mikkonen 2017).

Tässä testitapauksessa testattiin työaikapankin tuntien syöttämiseen käytettävää rest-rajapintaa http-patch -metodin avulla sekä syötettyjen tuntien hakemista rest-rajapinnasta http-get -metodia hyödyntäen. Testin on siis tarkoitus varmistaa, että rajapinta toimii ja tietoa käsitellään oikeassa muodossa. Rajapinnan on tarkoitus käsitellä tietoa JavaScript Object Notation (JSON) -muodossa. Testitapaus luettavaan muotoon kirjoitettuna näyttää tältä:

```
Testi 1. Tiedon syöttö rajapintaan patch-metodilla
1. Ota yhteys rajapintaan patch-metodilla
2. Annetaan haluttu JSON-objekti pyynnölle
3. Varmistetaan, että pyyntö onnistui (status 200)
4. Varmistetaan, että pyyntö vastaa syötetyllä objektilla

Testi 2. Tiedon hakeminen rajapinnasta get-metodilla
1. Ota yhteys rajapintaan get-metodilla
2. Varmistetaan, että pyyntö onnistui (status 200)
3. Varmistetaan, että pyyntö vastaa oletetulla objektilla
```

6.2 Rest-rajapinnan testaus Mocha-työkalulla

Testitapauksen luominen Mochalla aloitetaan asentamalla tarvittavat kirjastot. Testin luomiseen tarvitaan tietysti itse Mocha-kirjasto, kuin myös jokin todennuskirjasto, kuten kerrottiin opinnäytetyön kohdassa 5.1. Käytän testissä Mochan kanssa Chai-todennuskirjastoa. Lisäksi tarvitaan vielä Chain lisäosa, chai-http, joka mahdollistaa http-kutsujen käyttämisen yhdessä Mochan kanssa. Kirjastojen asentamista ja Mochan suorittamista varten täytyy suorittavassa laitteessa olla asennettuna Node.js sekä npm. Kirjastot asennetaan komentorivillä komennolla ”npm install mocha chai chai-http”.

Seuraavaksi täytyy luoda projektin juureen ”tests” kansio, josta Mocha oletuksena etsii suoritettavia testejä. Tähän kansioon luodaan sitten testitiedosto, annetaan sille nimeksi vaikkapa api.test.js. Vielä ennen itse testin kirjoittamista, sisällytetään kuitenkin tarvittavat kirjastot testitiedostoon (kuvio 10). Tässä testitapauksessa tarvitaan todennuskirjasto Chai ja sen todennusmetodit should ja expect, sekä chai-http http-pyyntöjen toteuttamista varten.

```
const chai = require('chai');
const chaiHttp = require('chai-http')
const should = chai.should();
const expect = chai.expect;
chai.use(chaiHttp);
```

Kuvio 10. Mocha-testissä tarvittavat kirjastot täytyy sisällyttää tiedoston alkuun, jotta niitä voi käyttää.

Testin kirjoittamisen luotuun testitiedostoon voi nyt aloittaa Mochan sekä Chain dokumentaatiota sekä suunniteltua testitapausta apuna käyttäen. Mocha-testitapaus koodiksi kirjoitettuna näyttää kuvion 11 mukaiselta.

src > tests > JS api.test.js > ...

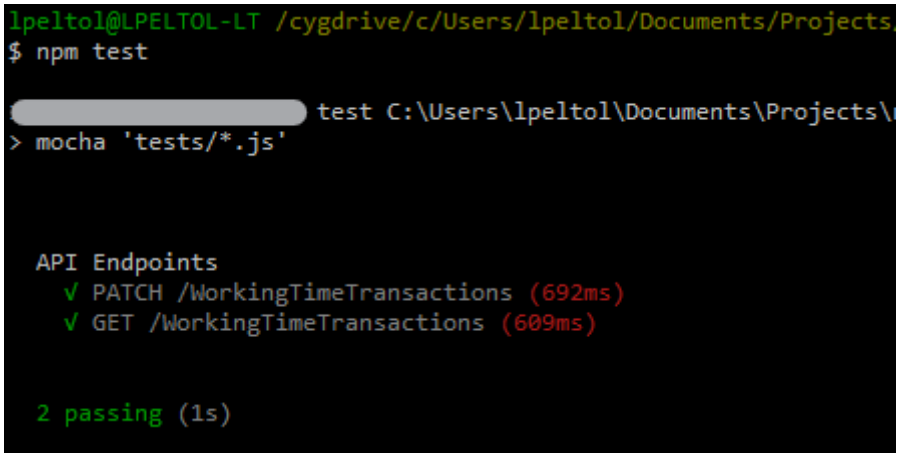
```
1   const chai = require('chai');
2   const chaiHttp = require('chai-http')
3   const should = chai.should();
4   const expect = chai.expect;
5   chai.use(chaiHttp);
6
7   const baseUrl = 'http[redacted]/workingtimebank';
8
9   describe('API Endpoints', () => {
10    it('PATCH /WorkingTimeTransactions', (done) => {
11      chai.request(baseUrl)
12        .patch('/WorkingTimeTransactions')
13        .send({
14          id: '1',
15          hour: 8,
16          ownerId: 'Testuser1'
17        })
18        .end((error, response) => {
19          response.should.have.status(200);
20          expect(response.body).to.deep.equal({
21            'id': '1',
22            'hour': 8,
23            'ownerId': 'Testuser1'
24          });
25          done();
26        });
27    });
28
29    it('GET /WorkingTimeTransactions', (done) => {
30      chai.request(baseUrl)
31        .get('/WorkingTimeTransactions')
32        .end((error, response) => {
33          response.should.have.status(200);
34          expect(response.body).to.deep.equal([
35            {
36              'id': '1',
37              'hour': 8,
38              'ownerId': 'Testuser1'
39            }
40          ]);
41        done();
42      });
43    });
44  });
45
```

Kuvio 11. Suunnitelman mukaan luotu Mocha-rajapintatesti.

Kuten kuvasta näkyy, testissä on rajapinnan kantaosoite, johon rajapintakutsu tehdään chai.request -metodin avulla. Mochassa asynkronisen koodin testaamiseen tarvitaan ainoastaan done-metodi, jolloin Mocha ymmärtää odottaa funktiokutsua ennen testisuorituksen päättämistä (Mocha 2020b). Ensimmäinen testi

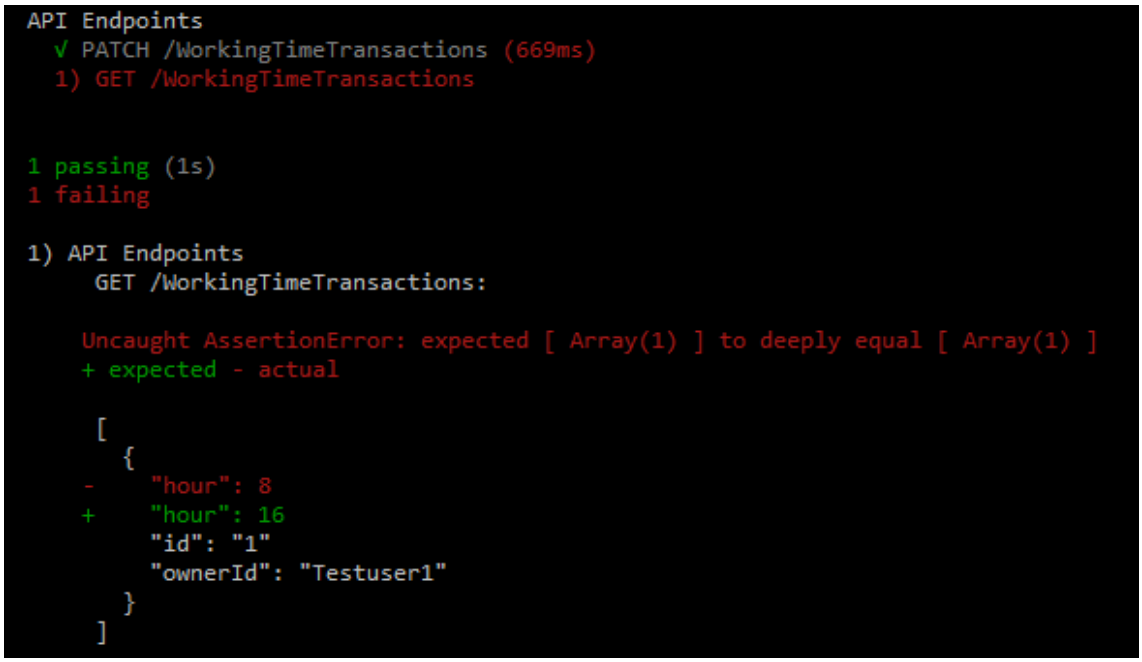
kutsuu ”/WorkingTimeTransactions” -rajapintaa http-patch -metodilla ja lähettää siihen JSON-objektin, jolla on ominaisuudet id = 1, hour = 8, ownerId = Testuser1. Tämän jälkeen testi varmistaa Chain should-metodilla, että pyynnön tila on 200, mikä tarkoittaa, että pyyntö onnistui. Lopuksi testi varmistaa vielä Chain expect-metodilla, että pyyntö vastaa oletetulla JSON-objektilla, joka on sama kuin pyynnössä lähetetty objekti. Toinen testi kutsuu rajapintaa http-get -metodilla. Tässä testissä yksinkertaisesti varmennetaan, että pyyntö onnistui ja että pyyntö vastaa odotetulla JSON-objektilla, jonka pitäisi olla ensimmäisessä testissä rajapintaan lähetetty objekti.

Seuraavaksi suoritetaan testi komentorivillä ajamalla komento ”npm test” (kuvio 12). Tämä komento on Mochan asennuksen yhteydessä automaattisesti luotu skripti, joka hakee ja suorittaa testit aiemmin luodusta tests-kansiosta.



Kuvio 12. Mocha-testien suorittaminen komentorivillä onnistui.

Testit menevät läpi, mutta tässä vaiheessa testien toiminnasta ei ole kuitenkaan muuta varmuutta, kuin se, että Mocha kertoo testien onnistuneen. Kokeilen vielä rikkoa testin vaihtamalla get-pyynnön oletetun vastauksen tarkoituksella vääräksi, jotta voin varmistua siitä, että rajapinta varmasti vastaa oikealla JSON-objektilla. Muutan hour-kentän vääräksi ja suoritan testin uudelleen (kuvio 13). Kuten kuvasta nähdään, testi epäonnistuu nyt. Mocha kertoo myös, että testi odotti hour-kentän arvon olevan 16, vaikka se todellisuudessa oli 8, kuten pitikin.



Kuvio 13. Mocha-testi epäonnistuu, koska rajapinta palauttaa erilaisen olion, kuin testissä on määritelty.

6.3 Rest-rajapinnan testaus Jest-työkalulla

Seuraavaksi toteutetaan täysin samanlainen testi Facebookin Jest-työkalun avulla. Aloitetaan testitapauksen luominen niin ikään asentamalla tarvittavat työkalut testin toteuttamista ja suorittamista varten. Jestin sanottiin olevan helppokäyttöinen, eikä se vaadikaan erillistä todennuskirjastoa Mochan tapaan. Jestin asennuksen mukana tulee nimittäin Expect.js -todennuskirjasto sisäänrakennettuna (Facebook 2020c). Http-pyynnön toteutusta varten Jest tarvitsee kuitenkin erillisen kirjaston avukseen, ellei pyyntöä halua toteuttaa mock-pyyntönä (Facebook 2020d). Käytän tässä tapauksessa Supertest-kirjastoa, jonka avulla http-pyyntöjä voi toteuttaa testitapauksia varten (Supertest 2019). Myös Jestin suorittamiseen ja kirjastojen asentamiseen tarvitsee Mochan tavoin suorittavassa laitteessa olla Node.js sekä npm asennettuna. Asennan tarvittavat kaksi kirjastoa komentoriviltä komennolla ”npm install jest supertest”.

Tämän enempää konfiguraatiota Jestin testausympäristön pystyttämiseksi ei vaadita. Luon seuraavaksi kansion nimeltä ”__tests__” Jest-testiä varten. Kansion voi nimetä miten haluaa, sillä Jestin asennuksen mukana tuleva testit

suorittava skripti osaa hakea testit projektin kansioista. Tätä nimeämiskäytäntöä on kuitenkin seurattu niin Jestin dokumentaatiossa, kuin toimeksiantajan olemassa olevissa Jest-testeissäkin, joten käytän myös tässä testissä samaa käytäntöä. Luon kansioon uuden tiedoston, jolle annan nimeksi jestApi.test.js. Mochasta poiketen Jest-testitiedostoon ei tarvitse sisällyttää mitään kirjastoja, koska todennuskirjasto Expect on sisäänrakennettuna. Tähän testiin tarvitaan kuitenkin Supertest-kirjasto, joten sisällytän sen testin alkuun. Kokonaisuudessaan kirjoitettu Jest-testi on kuvattu kuviossa 14.

src > __tests__ > JS jestApi.test.js > ...

```
1   const request = require('supertest')
2
3   const baseUrl = 'htt                                    workingtimebank';
4
5   describe('API Endpoints', () => {
6     it('PATCH /WorkingTimeTransactions', async () => {
7       const res = await request(baseUrl)
8         .patch('/WorkingTimeTransactions')
9         .send({
10          id: '2',
11          hour: 14,
12          ownerId: 'Testuser2'
13        });
14      expect(res.statusCode).toEqual(200);
15      expect(res.body).toEqual({
16          'id': '2',
17          'hour': 14,
18          'ownerId': 'Testuser2'
19      });
20    });
21
22    it('GET /WorkingTimeTransactions', async () => {
23        const res = await request (baseUrl)
24          .get('/WorkingTimeTransactions');
25        expect(res.statusCode).toEqual(200)
26        expect(res.body).toEqual([
27          {
28            'id': '2',
29            'hour': 14,
30            'ownerId': 'admin'
31          }
32        ]);
33    });
34  });
35
```

Kuvio 14. Suunnitelman mukaan kirjoitettu Jest-testi.

Kuten testitapauksesta näkyy, testi muistuttaa pitkälti Mochalla ja Chailla tehtyä testiä. Joitakin eroja kuitenkin on, esimerkiksi kirjastoja ei testiin tarvitse sisällyttää niin paljoa, kuin Chaita käyttäessä. Jestissä http-pyyntöön täytyy lisätä esimerkiksi async ja await -avainsanat, jotta Jest osaa odottaa asynkronisen funktiokutsun päättymistä samaan tapaan, kuin Mocha open-metodia käytettäessä (Facebook 2020e). Ensimmäinen testi kutsuu ”/WorkingTimeTransactions” -rajapintaa http-patch -metodilla, ja lähettää siihen JSON-objektin, jolla on ominaisuudet id = 2, hour = 14, ownerId = Testuser2. Testi varmistaa, että pyyntö onnistuu (status 200) ja että pyyntö vastaa oletetulla JSON-objektilla. Toinen testi kutsuu rajapintaa http-get -metodilla ja varmistaa, että kutsu onnistuu ja palauttaa oikean objektin. Varmistukseen käytetään Expect.js -kirjaston toEqual-metodia.

Testin voi suorittaa komentorivillä ajamalla komento ”npm test”. Tämä on Mochan tavoin testaustyökalun asennuksen mukana tuleva skripti, joka hakee ja suorittaa Jest-testit projektista. On kuitenkin syytä huomioida, että jos projektiin on asennettu sekä Mocha että Jest, tämä skripti käynnistää vain ensin asennetun testaustyökalun. Jestin voi käynnistää myös suorittamalla sen asennuskansiosta, joka on projektin ”node_modules” -kansion sisällä. Suoritetaan Jest ajamalla komento ”node_modules/.bin/jest” (kuvio 15).

```
lpeltol@LPELTOL-LT /cygdrive/c/Users/lpeltol/Documents/Projects/
$ node_modules/.bin/jest
 PASS  src/__tests__/jestApi.test.js
  API Endpoints
    √ PATCH /WorkingTimeTransactions (667ms)
    √ GET /WorkingTimeTransactions (571ms)

Test Suites: 1 passed, 1 total
Tests:       2 passed, 2 total
Snapshots:   0 total
Time:        4.027s
Ran all test suites.
```

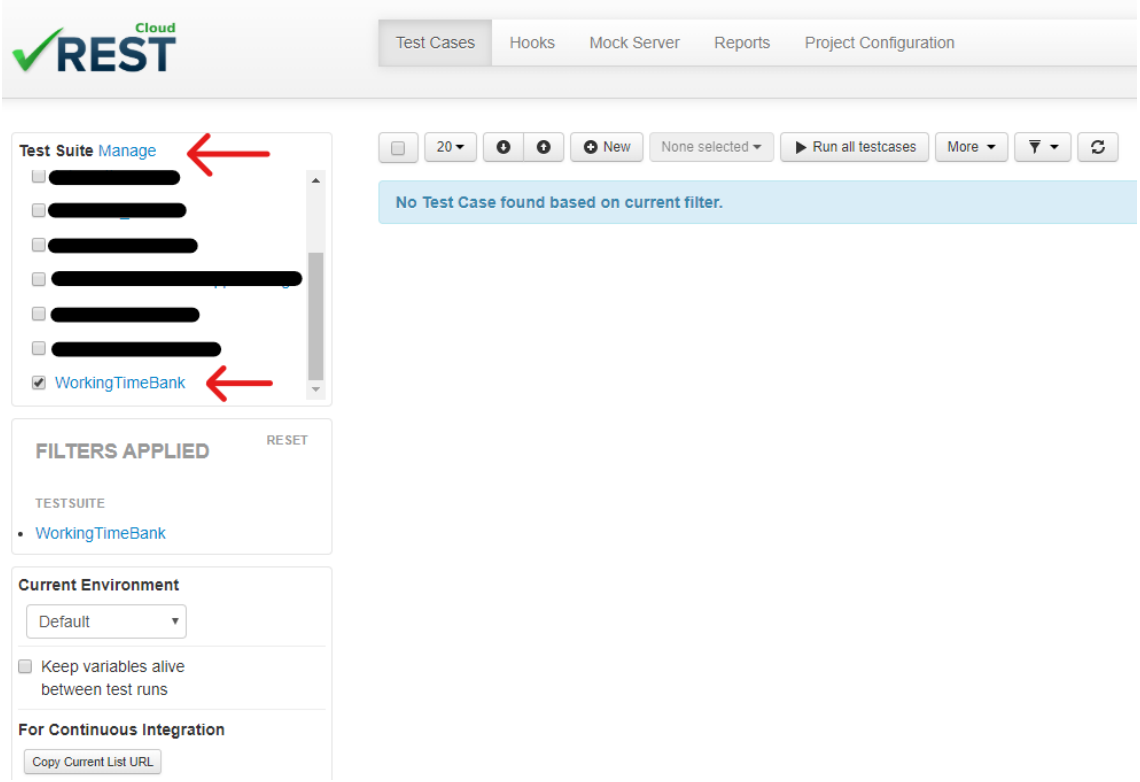
Kuvio 15. Jest-testit menevät läpi.

Rikotaan testi vielä samalla tavalla kuin Mocha-testitapauksessa, jotta voidaan varmistua, että kaikki toimii kuten pitääkin. Vaihdan ownerId-kentän arvoksi testiin ”admin” ja suoritan testin uudelleen. Testi epäonnistuu ja Jest kertoo, että odotettu arvo on eri, kuin rajapinnan palauttama arvo (kuvio 16). Tämä tarkoittaa, että rajapinta palauttaa sen mitä pitääkin.

Kuvio 16. Jest ilmoittaa, että odotettu arvo on eri, kuin rajapinnan palauttama arvo ja testi epäonnistuu.

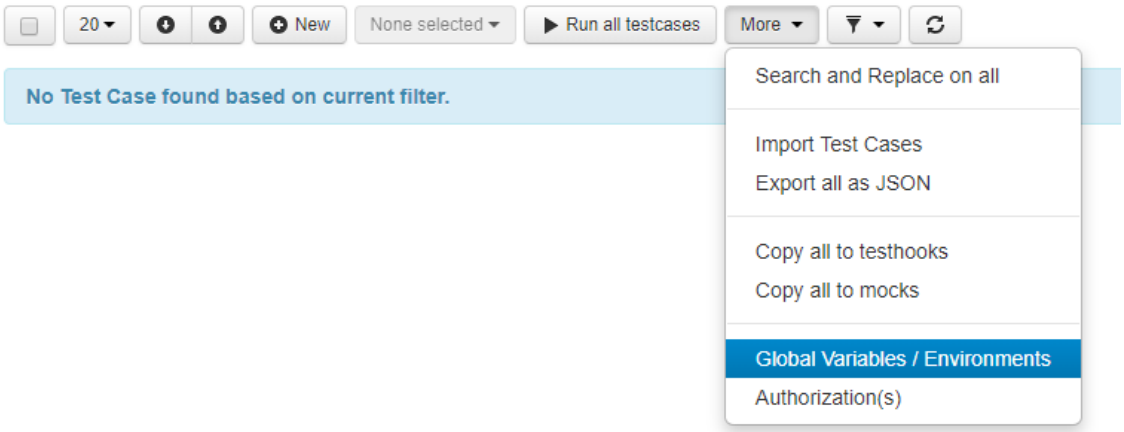
6.4 Rest-rajapinnan testaus vREST-työkalulla

Testitapauksen luominen vREST-työkalulla aloitetaan luomalla uusi testiympäristö (Test Suite) vREST-pilvipalveluun. Uusi ympäristö luodaan valitsemalla käyttöliittymästä ”Manage” ja syöttämällä sitten ympäristölle haluttu nimi (kuvio 17). Annan testiympäristölle nimeksi projektin nimen, WorkingTimeBank.



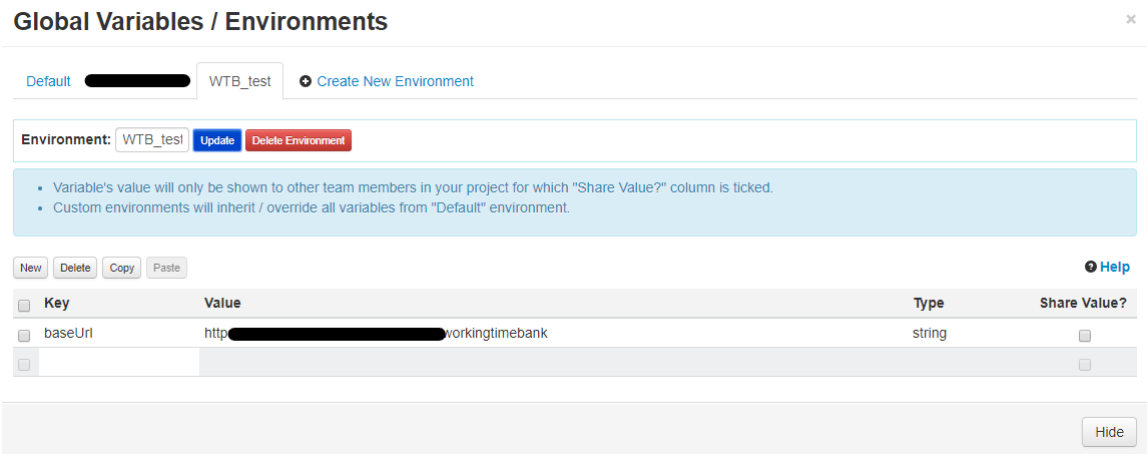
Kuvio 17. Testiympäristön luominen vREST-pilvipalvelussa.

Muuta konfiguraatiota ei tarvita, mutta lisään vielä rajapinnan kantaosoitteen testiympäristön ympäristömuuttujiin valitsemalla käyttöliittymästä ”Global Variables / Environments” (kuvio 18).



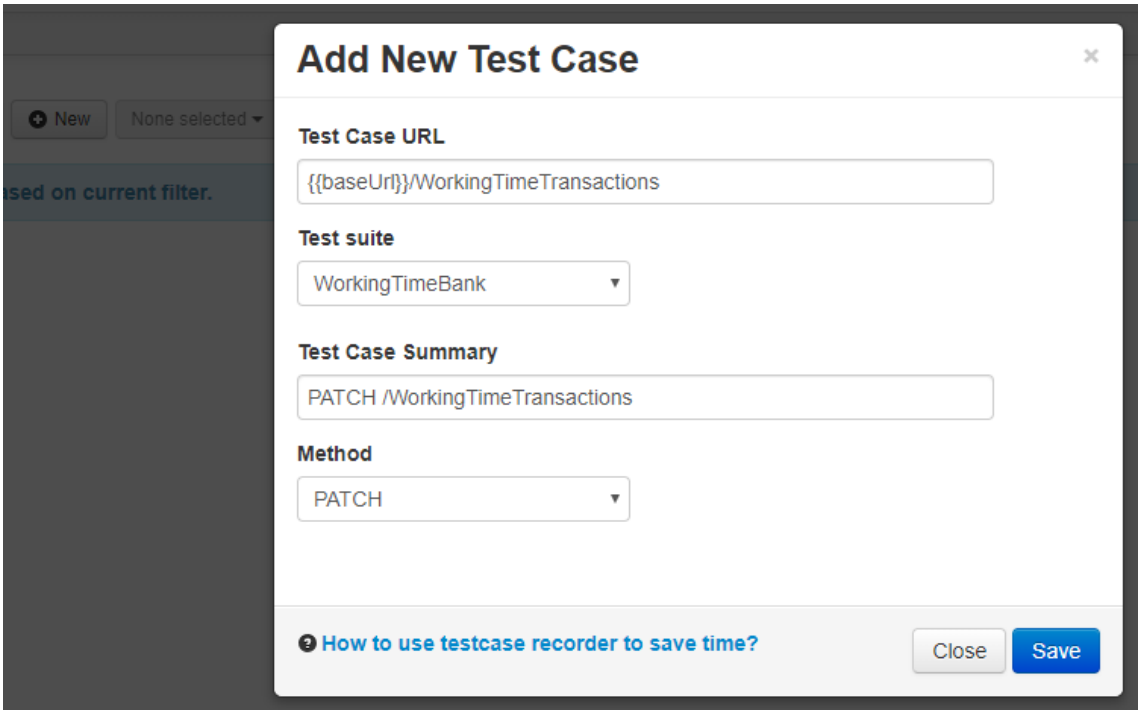
Kuvio 18. Muuttujien ja ympäristöjen lisääminen vREST-palveluun.

Luon uuden ympäristön testin ympäristömuuttujia varten ja lisään sinne baseUrl-muuttujan (kuvio 19).



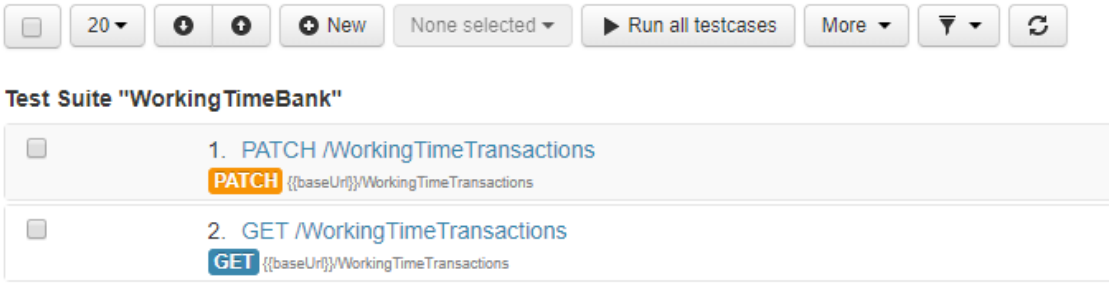
Kuvio 19. Kantaosoite lisättynä testin ympäristömuuttujaksi.

Nyt kaikki konfiguraatio on tehty ja testitapauksen voi luoda palveluun. Uusi testitapaus luodaan valitsemalla käyttöliittymästä ”New”. Testiin tulee syöttää osoite, testausympäristö, kuvaus sekä käytettävä http-pyyntö. Ensimmäiseen testiin syötän suunnitelman mukaisesti työaikapankin rest-rajapinnan osoitteen, käyttäen apuna juuri lisättyä kantaosoite-muuttujaa. Testausympäristönä käytän aiemmin luotua WorkingTimeBank -ympäristöä. Valitsen http-pyynnön tyypiksi patch, ja annan testille vielä kuvauksen (kuvio 20).



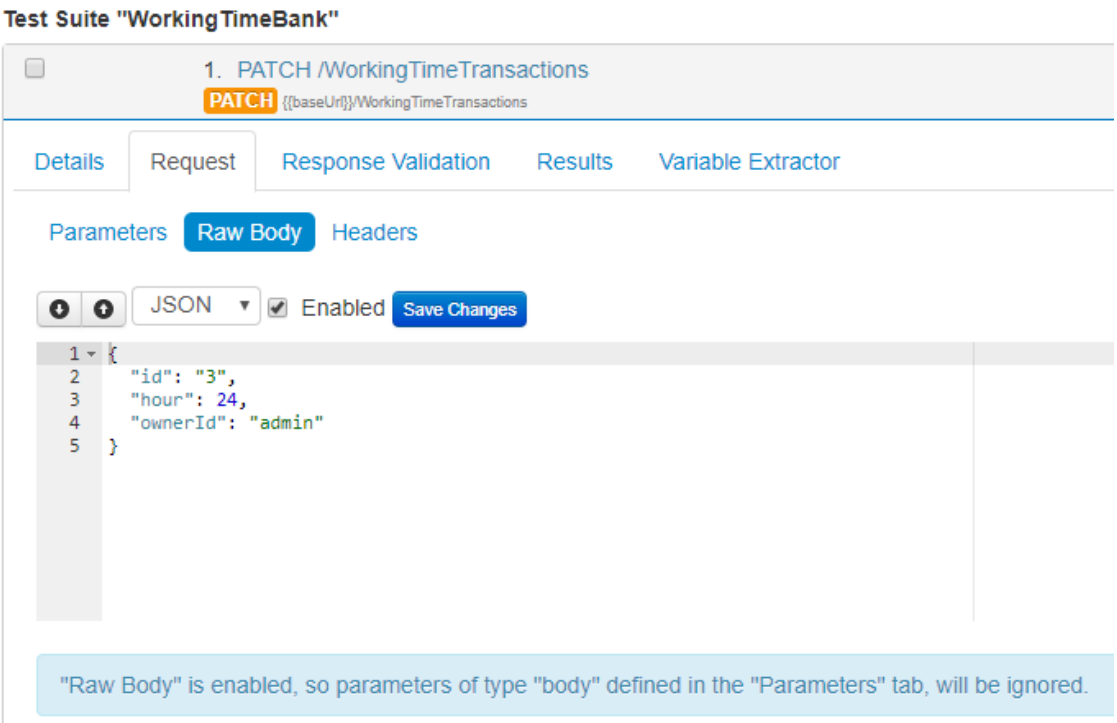
Kuvio 20. Testitapauksen lisääminen vREST-palveluun.

Luon samalla tavalla vielä toisen testin, joka testaa rajapintaa http-get -metodilla. Testit näkyvät nyt testiympäristön käyttöliittymässä ja niitä voi muokata ja suorittaa (kuvio 21).



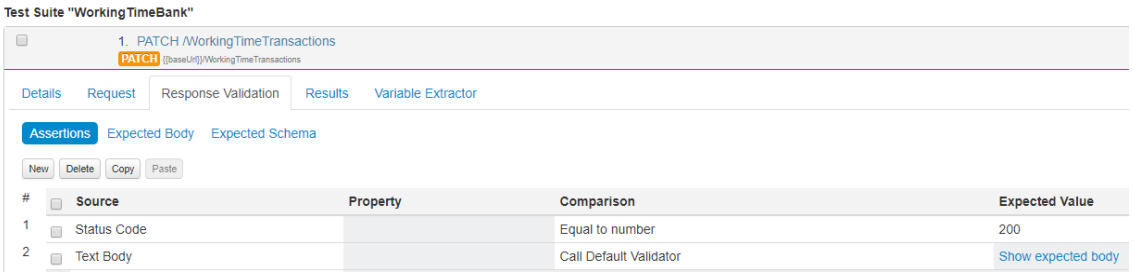
Kuvio 21. Luodut testitapaukset vREST-palvelussa.

Nyt testin voi konfiguroida haluamallaan tavalla. Ensimmäisessä testissä tietoa halutaan lähettää, joten lisään sen pyyntöön JSON-objektin, jonka haluan kutsussa lähettää. Pyynnön mukana lähetettävä data voidaan lisätä testin ”Request” -välilehdestä. Datan voi määritellä joko parametreinä tai raakana JSON-datana (kuvio 22).



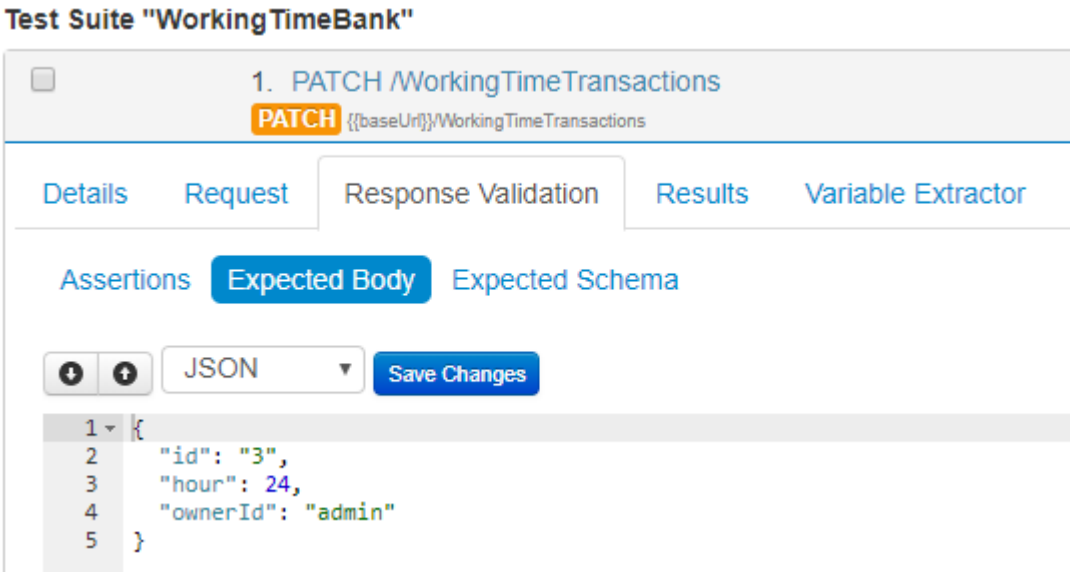
Kuvio 22. Patch-pyynnön määrittäminen vREST-testissä.

Todennukset lisätään testin ”Response Validation” -välilehdestä. Testisuunnitelmassa määritettiin, että testin pitää vastata statuskoodilla 200, sekä varmistaa, että pyynnön palauttama data on sama, kuin lähetetty data. Lisään testiin kaksi todennusta, jotka todentavat nämä kohdat (kuvio 23).



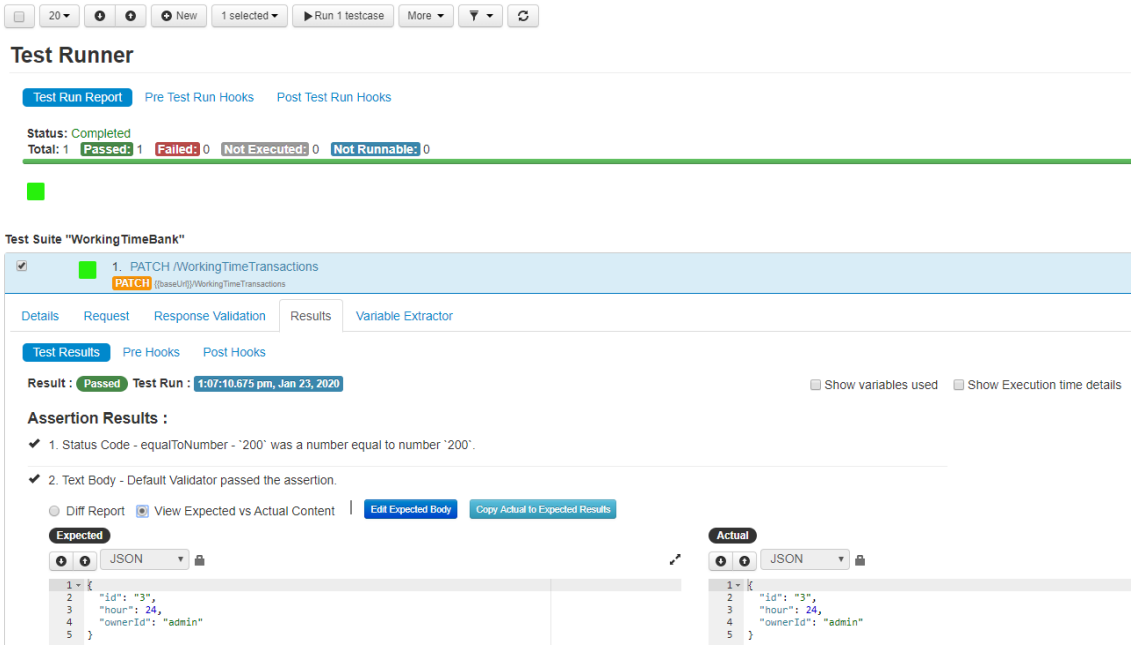
Kuvio 23. Todennusten lisääminen vREST-testiin.

Määritän vielä ”Text Body” -todennusmuuttujan arvoksi sen JSON-objektin, jonka odotan rajapinnan palauttavan (kuvio 24).



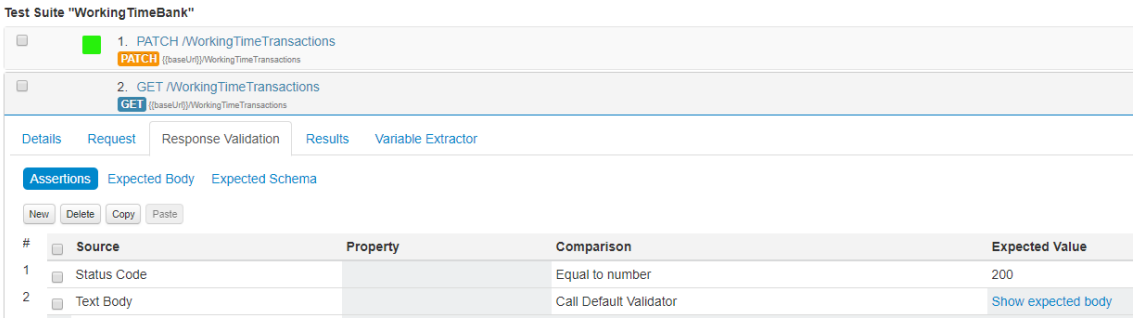
Kuvio 24. Oletetun palautusarvon määrittäminen vREST-testissä.

Nyt kun http-kutsu ja suunnitelman mukaiset todennukset on määritetty, testin voi ajaa käyttöliittymästä. Testi onnistuu ja käyttöliittymästä voi tarkastella tuloksia (kuvio 25).



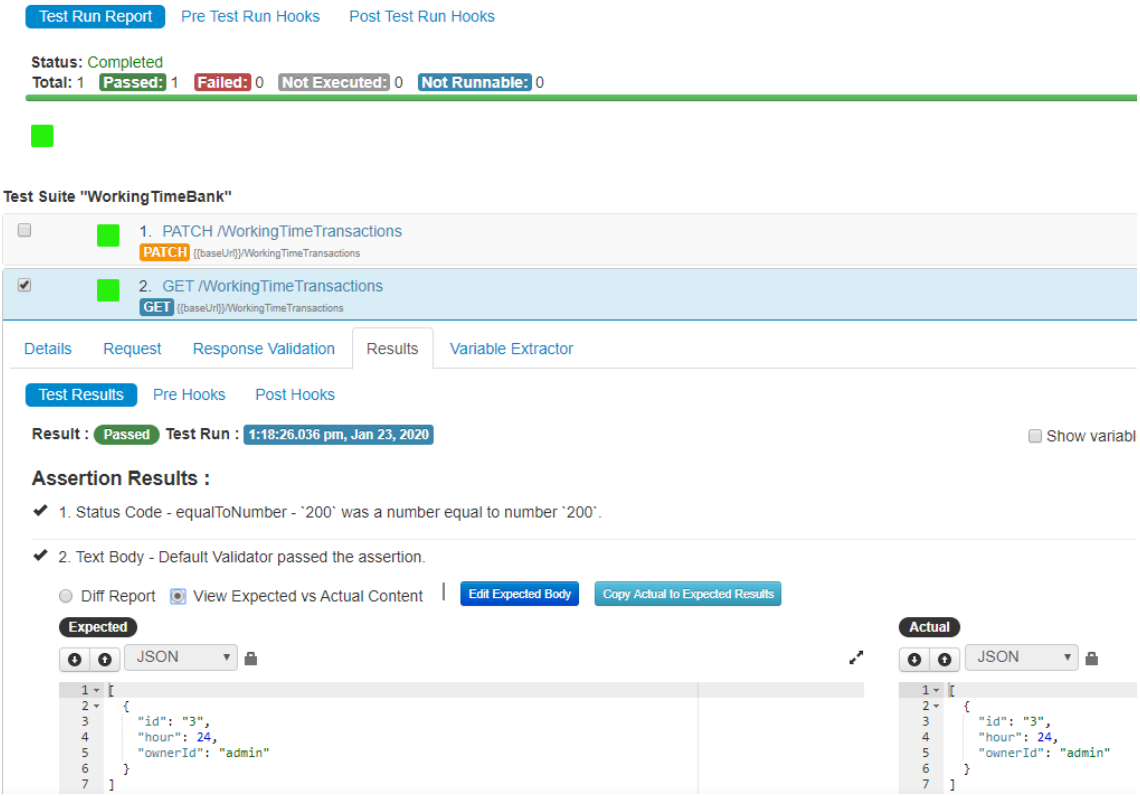
Kuvio 25. Onnistuneesti suoritetun testin tulokset vREST-pilvipalvelussa.

Suoritetaan sitten vielä toinen testi, joka kutsuu rajapintaa http-get -metodilla ja todentaa pyynnön onnistuneeksi sekä palautusarvon oikeaksi. Määritetään todennukset samalla tavalla, kuin patch-testissä (kuvio 26).



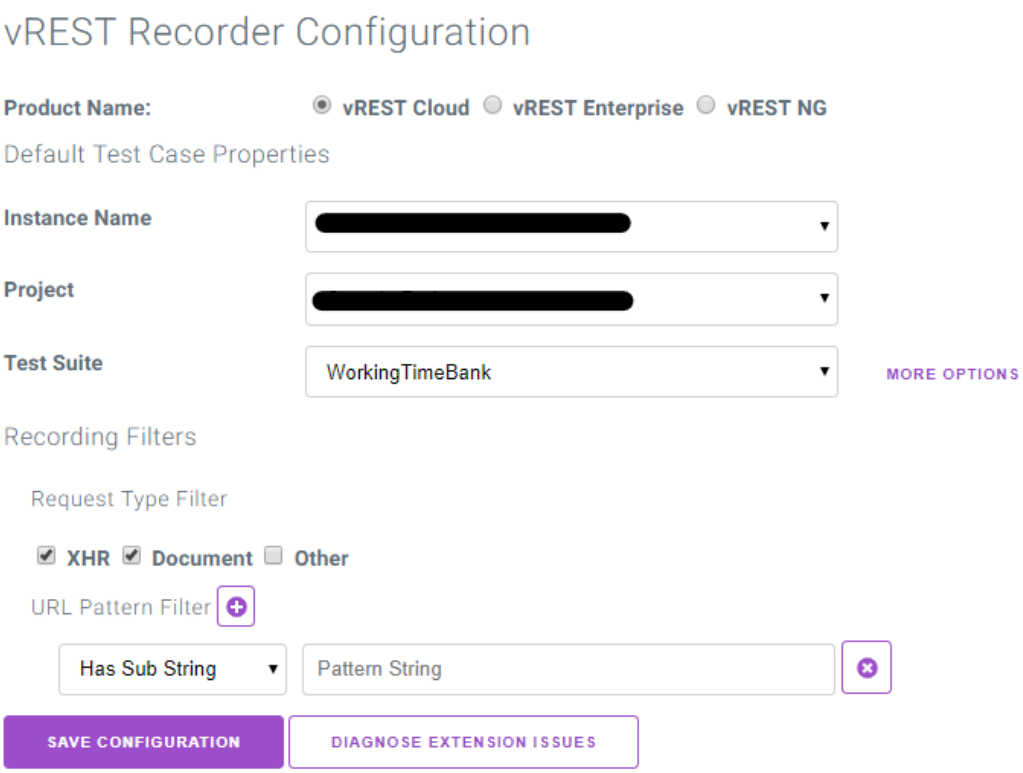
Kuvio 26. Todennusten määrittäminen toista testiä varten.

Toisenkin testin voi nyt suorittaa. Testi menee läpi onnistuneesti ja tuloksista nähdään, että odotetut arvot vastasivat todellisia (kuvio 26).



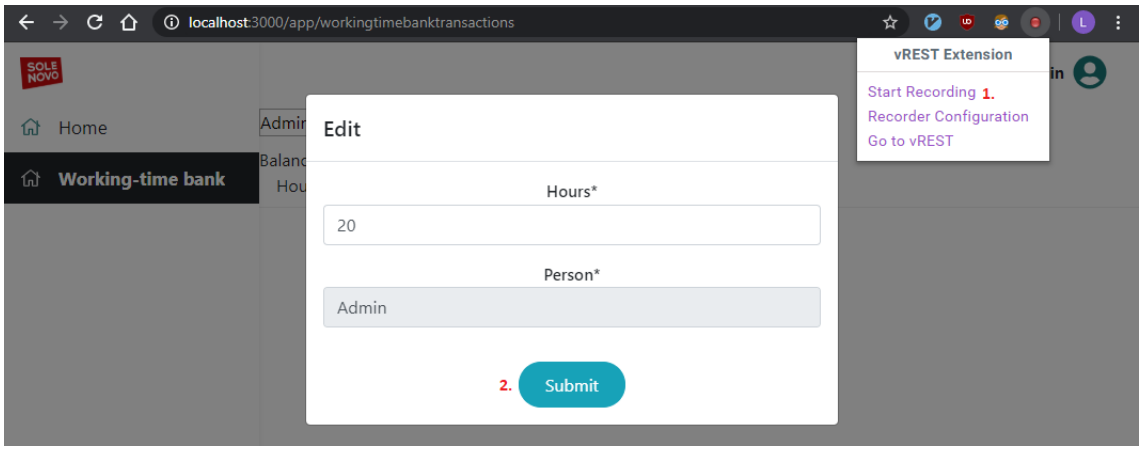
Kuvio 27. Rajapinta vastaa http-get -pyyntöön onnistuneesti.

Yksi vREST-palvelun keskeisistä ominaisuuksista on luvussa 5.3 mainittu testien nauhoittaminen. Testien nauhoittaminen onnistuu asentamalla ”vREST API Testing Tool” -laajennus Chrome-selaimeen. Kun laajennus on asennettu, pääsee sen asetuksiin selaimen yläpalkista. Käyn valitsemassa asetuksista aiemmin luodun WorkingTimeBank -testausympäristön, jotta nauhoitetut testit tallentuvat oikeaan paikkaan (kuvio 28).



Kuvio 28. vREST-nauhoitustyökalun määrittäminen.

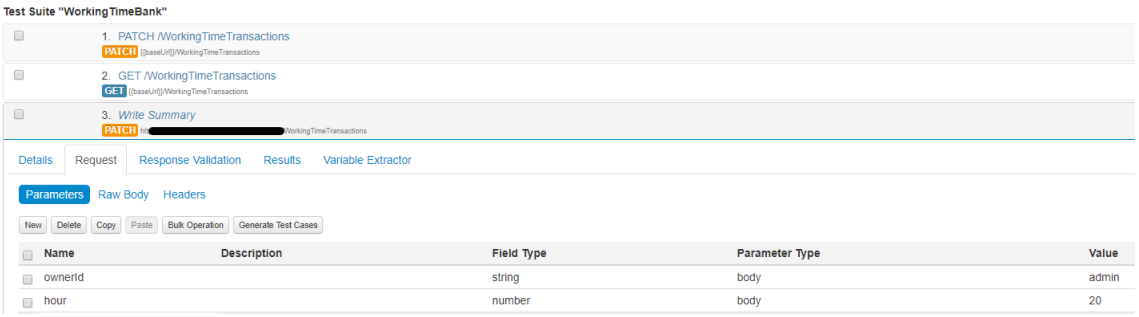
Tämän jälkeen voin luoda testitapauksen automaattisesti käynnistämällä nauhoituksen ja käyttämällä työaikapankki-sovellusta. Haluan testata tuntien lisäämiseen käytettävää rajapintaa, joten navigoin sovellukseen ja syötän tunnit, mutta ennen pyynnön lähettämistä käynnistän vREST-nauhoituksen (kuvio 29).



Kuvio 29. Nauhoituksen aloittaminen vREST API Testing -työkalulla.

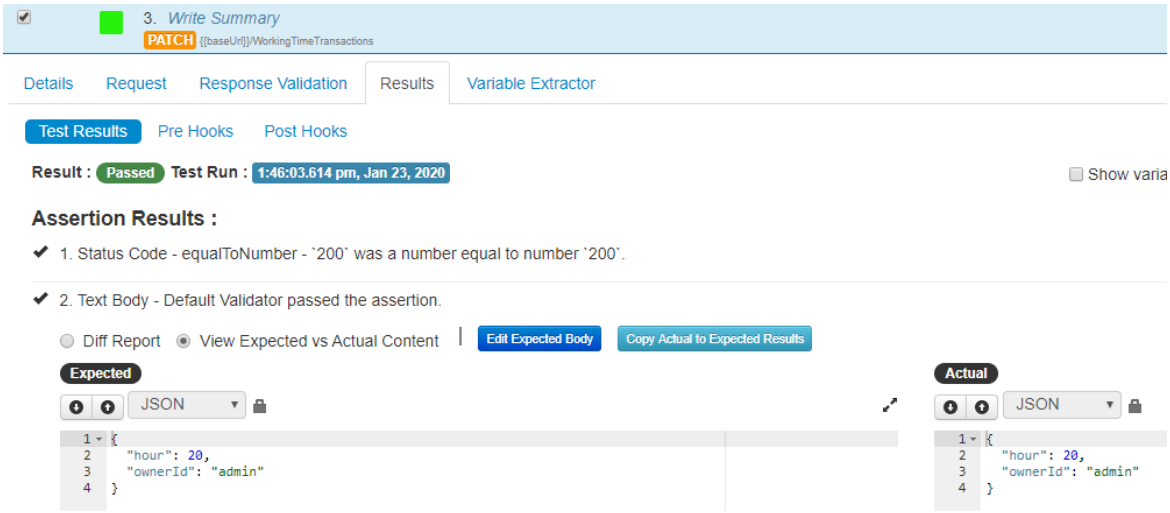
Kun tunnit on lähetetty, pysäytän nauhoituksen laajennuksesta ja päivitän vREST-pilvipalvelun testisivun. Uusi testi on automaattisesti ilmestynyt

ympäristöön, ja sitä tarkastelemalla huomataan, että siinä on myös käyttöliittymän kautta lähetetyt tiedot parametreinä (kuvio 30).



Kuvio 30. Nauhoitustyökalu luo automaattisesti uuden testitapauksen vREST-pilvipalveluun.

Muutan testin osoitteen vielä käyttämään testiympäristön kantaosoite-ympäristömuuttujaa. Testiin voi myös lisätä nyt halutut todennukset, jonka jälkeen testin voi suorittaa (kuvio 31).



Kuvio 31. Nauhoitustyökalun avulla luodun testin suorittaminen onnistuu.

Kuten kuvasta näkyy, automaattisesti luodun testin suorittaminen onnistuu. Tässä testitapauksessa ei ole muista testeistä poiketen id-kenttää, koska käyttöliittymän ei käytä sitä kutsuessaan rajapintaa.

7 React.js testauskehykset

7.1 Selenium

Selenium on avoimen lähdekoodin ilmainen verkkosivujen automaatiotestaukseen tarkoitettu kokoelma työkaluja. Selenium-tuoteperhe muodostuu kolmesta työkalusta, jotka ovat Selenium Webdriver, Selenium IDE (Integrated Development Environment) sekä Selenium Grid. (Selenium 2020a.) Webdriver on rajapinta, joka mahdollistaa selaimen automaattisen käytön paikallisesti tai etänä. Webdriverin avulla selaimessa toimivan sovelluksen voi automatisoida käyttäytymään siten, kuin oikea käyttäjä käyttäisi sitä. (Selenium 2020b.) Selenium IDE on selaimeen asennettava lisäosa ja se on saataville Chrome ja Firefox -selaimille. Selenium IDE:n tarkoitus on olla helppokäyttöinen, eikä se vaadi käyttäjältä asennuksen lisäksi mitään konfigurointia. Selenium IDE mahdollistaa koko testitapauksen luomisen yhdellä työkalulla. Työkalun avulla käyttäjä voi nauhoittaa verkkosovelluksen toimintaa klikkailemalla sivua ja IDE tallentaa toiminnot ketjuksi. Näin muodostuu testitapaus, jonka voi tallentaa myöhempää käyttöä varten. (Selenium 2019a.) Selenium Webdriverin ja IDE:n lisäksi Selenium tarjoaa vielä Grid-työkalun. Selenium Grid mahdollistaa testien yhtäaikaisen ajamisen erilaisilla selain- ja laitekombinaatioilla. (Selenium 2019b.)

Tämän opinnäytetyön Selenium-testitapauksessa käytettiin Selenium IDE:ä yhdessä sen lisäosan, Selenium side-runnerin, kanssa. Selenium side-runner mahdollistaa Selenium IDE:llä nauhoitettujen testien suorittamisen komentorivillä (Selenium 2019c). Selenium valikoitui yhdeksi opinnäytetyön käyttöliittymätestaustyökaluista, koska testien toteuttaminen sillä on pitkälti samanlaista kuin Screenerillä, joka on jo yrityksellä käytössä. Lisäksi myös Selenium on ollut yrityksellä käytössä joissakin aiemmissa projekteissa ja opinnäytetyö tarjosi hyvän tilaisuuden selvittää vastaako Selenium vanhahkona työkaluna modernien verkkosovellusten testaustarpeisiin.

7.2 Screener

Screener on automatisoituun visuaaliseen testaukseen tarkoitettu maksullinen työkalu. Screener tarjoaa kaksi erillistä palvelua: yksittäisten komponenttien testaukseen tarkoitetun Screener Componentsin, jota voi käyttää esimerkiksi React- tai Vue -komponenttien testaukseen, sekä verkkosovelluksen end-to-end testaukseen tarkoitetun Screener E2E:n. (Screener 2020a).

Tässä opinnäytetyössä käytettiin Screener E2E-testaustyökalua. Screener E2E:n käyttö oli hyvin samankaltaista, kuin Selenium IDE:n. Testejä voidaan nauhoittaa Screener Recorder nimisellä selaimeen asennettavalla lisäosalla. Lisäosa on saatavilla ainoastaan Chrome-selaimelle (Screener 2020b). Screener E2E -testit toimivat myös käytännössä samalla tavalla, kuin Seleniumissa. Selain navigoi automaattisesti verkkosovelluksessa käyttäjän määrittämää reittiä pitkin esimerkiksi klikkailemalla sivun elementtejä ja linkkejä. Screener E2E:n tärkein ominaisuus on ottaa halutuista vaiheista kuvakaappauksia. Screenerin testit suoritetaan yrityksen omassa pilvipalvelussa, jossa järjestelmä vertailee jokaisen testin suorituskerran jälkeen otettuja kuvakaappauksia. Mikäli sivuston ulkonäkö on muuttunut, siitä ilmoitetaan käyttäjälle. Käyttäjä voi tarkastelun jälkeen hyväksyä tai hylätä muutokset. (Screener 2020c.)

Screener E2E valikoitui opinnäytetyöhön, koska se on toimeksiantajalla käytössä jo muissakin projekteissa. Oli myös mielenkiintoista nähdä, miten kaupallisen testaustyökalun kanssa työskentely eroaa avoimen lähdekoodin työkaluista. Etenkin eroavaisuudet Seleniumin kanssa, jonka käyttö on niin samankaltaista, kuin Screenerin, kiinnostivat. Screenerin voi myös integroida CI/CD -putkeen, jolloin palvelu suorittaa testit automaattisesti kehityksen halutussa vaiheessa (Screener 2020d).

7.3 Cypress

Cypress on verkkosovellusten käyttöliittymien testaukseen tarkoitettu työkalu. Cypress tarjoaa kaksi tuotetta: Cypress Test Runner sekä Cypress Dashboard

Service. Cypress Test Runner on ilmainen ja avoimen lähdekoodin ohjelmisto, kun taas Dashboard-version hinta vaihtelee ilmaisesta maksulliseen, riippuen muun muassa käyttäjien sekä testien määrästä. Dashboard-versio on web-käyttöliittymä, joka tarjoaa käyttäjälle enemmän tietoa testien suorituksesta. Testit itsessään voidaan kuitenkin toteuttaa kokonaan Test Runnerilla (Cypress 2020a.) Koska pelkkä Test Runner riittää itse testien toteuttamiseen ja suorittamiseen, tämän opinnäytetyön Cypress-testaus osiossa tullaan käyttämään Cypress Test Runneria. Tästä eteenpäin, kun tässä työssä puhutaan Cypressistä, viitataan sillä Cypress Test Runner -työkaluun.

Cypress kattaa web-testauksen kaikki tasot. Sillä voidaan toteuttaa yksikkö-, integraatio- sekä E2E-testejä. Cypressin avulla voi testata mitä tahansa selaimessa käytettävää sovellusta. Seleniumista ja Screeneristä poiketen Cypressin testit toteutetaan käsin ohjelmakoodia kirjoittamalla eikä selaimen toimintaa nauhoittamalla. Muuten Cypressillä toteutettu E2E-testit toimivat kuten Seleniumissa ja Screenerissäkin. Työkalu pyörittää automaattisesti selainta, joka seuraa käyttäjän määrittämää reittiä ja toteuttaa käyttäjän määräämiä toimintoja. (Cypress 2020b.) Cypress valittiin tämän opinnäytetyön kolmanneksi käyttöliittymätestaustyökaluksi, koska se on uusi ja nopeasti suurta suosiota saanut kirjasto, eikä se ole toimeksiantajan projekteissa tällä hetkellä vielä käytössä. Vaikka Cypressin käyttö hieman eroaakin Seleniumista ja Screeneristä, voi sillä kuitenkin toteuttaa täysin samanlaisia testejä. Tämä on toinen syy, miksi Cypress haluttiin ottaa mukaan tähän opinnäyteyöhön.

8 React.js käyttöliittymän testaus

8.1 Testitapaus

Käyttöliittymätesti tulee suunnitella aina loppukäyttäjän näkökulmasta. Testin suunnittelussa täytyy siis miettiä, miten käyttäjä liikkuu sovelluksessa ja millaisia toimintoja hän sovelluksessa käyttää. (Software Testing Help 2019b.) Koska testatussa työaikapankkisovelluksessa ei ole varsinaisesti muuta toimintoa kuin

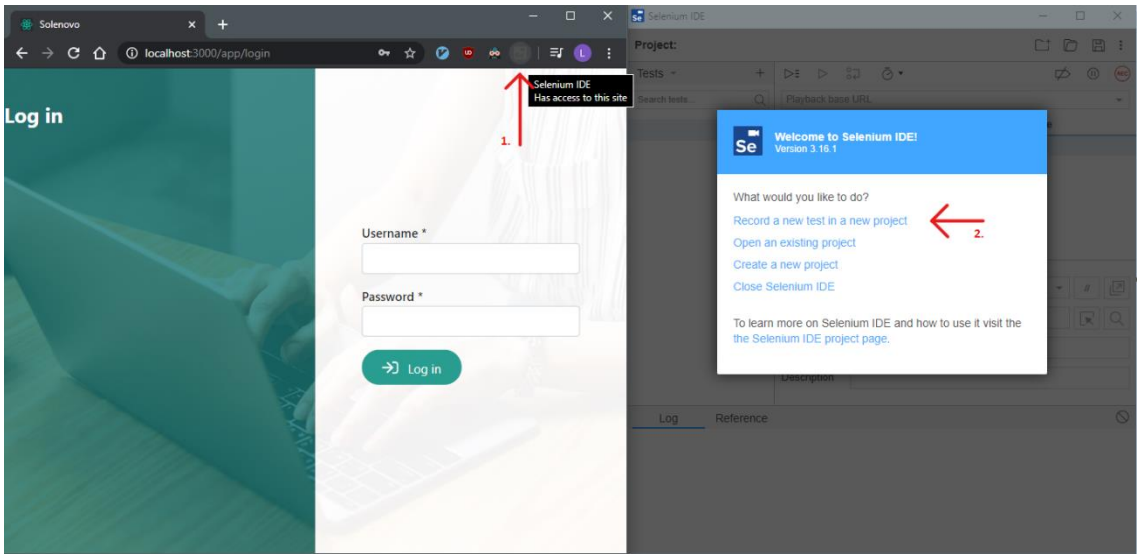
tuntien kirjaaminen, testattiin tämän osion testitapauksessa sitä. Kun testitapauksia tulee tässä vaiheessa vain yksi, sisällytettiin siihen myös käyttäjän sisäänkirjautuminen. Jos testejä tehtäisiin tulevaisuudessa useampia, kirjautuminen pitäisi pystyä ohittamaan siten, ettei sitä tarvitse tehdä jokaisen testin yhteydessä uudestaan. Luettavaan muotoon kirjoitettuna testitapaus näyttää tältä:

```
1. Avaa verkkosivu
2. Hae sähköpostikenttä -> kirjoita kenttään sähköpostiosoite
3. Hae salasanakenttä -> kirjoita kenttään salasana
4. Hae sisäänkirjautumispainike -> klikkaa painiketta
5. Kirjautuminen onnistuu ja sovellus siirtyy etusivulle
6. Hae valikosta työaikapankki-sivun linkki -> klikkaa linkkiä
7. Sovellus siirtyy työaikapankki-sivulle
8. Hae käyttäjävalikko -> vaihda käyttäjää valikosta
9. Hae painike, jolla lisätään tunteja -> klikkaa painiketta
11. Tuntien syöttöikkuna aukeaa
12. Hae tuntien syöttökenttä -> syötä kenttään tunteja
13. Hae lisää tunnit -painike -> klikkaa painiketta
14. Tuntien syöttöikkuna sulkeutuu
15. Varmista, että syötetyt tunnit näkyvät sivulla -> hae tunnit kentästä syötettyjä tunteja
```

Testin tarkoitus on matkia sitä reittiä, mitä tavallinen käyttäjä sivustolla kulkisi, ja niitä toimintoja, joita hän sivustolla käyttäisi. Yksinkertaistettuna käyttäjä haluaa kirjautua sisään, navigoida työaikapankkiin ja lisätä tunteja onnistuneesti haluamalleen käyttäjälle. Testitapauksessa oletetaan, että valitulla käyttäjällä ei ole ennestään pankkiin kirjattuja tunteja.

8.2 Käyttöliittymätestin toteuttaminen Selenium IDE -työkalulla

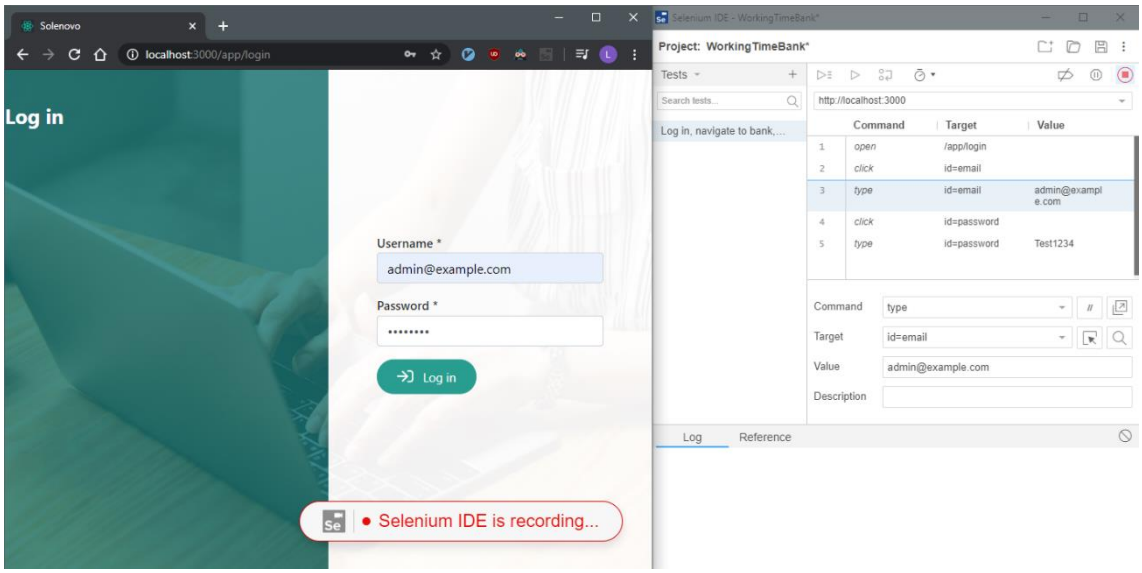
Testitapauksen luominen alkaa asentamalla Selenium IDE -lisäosa Chrome-selaimeen. Asennuksen jälkeen työkalu aukeaa Chromen yläpalkista. IDE:n aloitusikkunasta valitaan uuden projektin luominen (kuvio 32).



Kuvio 32. Selenium IDE:n avaus Chrome-selaimessa ja uuden projektin luominen.

Projektin luomisen yhteydessä IDE pyytää käyttäjää antamaan projektille nimen sekä kantaosoitteen, jota vasten testejä toteutetaan. Tässä tapauksessa annetaan projektille nimeksi sovelluksen nimi, joka on WorkingTimeBank. Kantaosoitteeksi asetetaan paikallinen osoite, jossa sovellusta testauksen yhteydessä ajetaan, eli http://localhost:3000. Jos tuote olisi jo julkaistu, voisi tähän asettaa verkkosivun oikean osoitteen, kuten www.example.com.

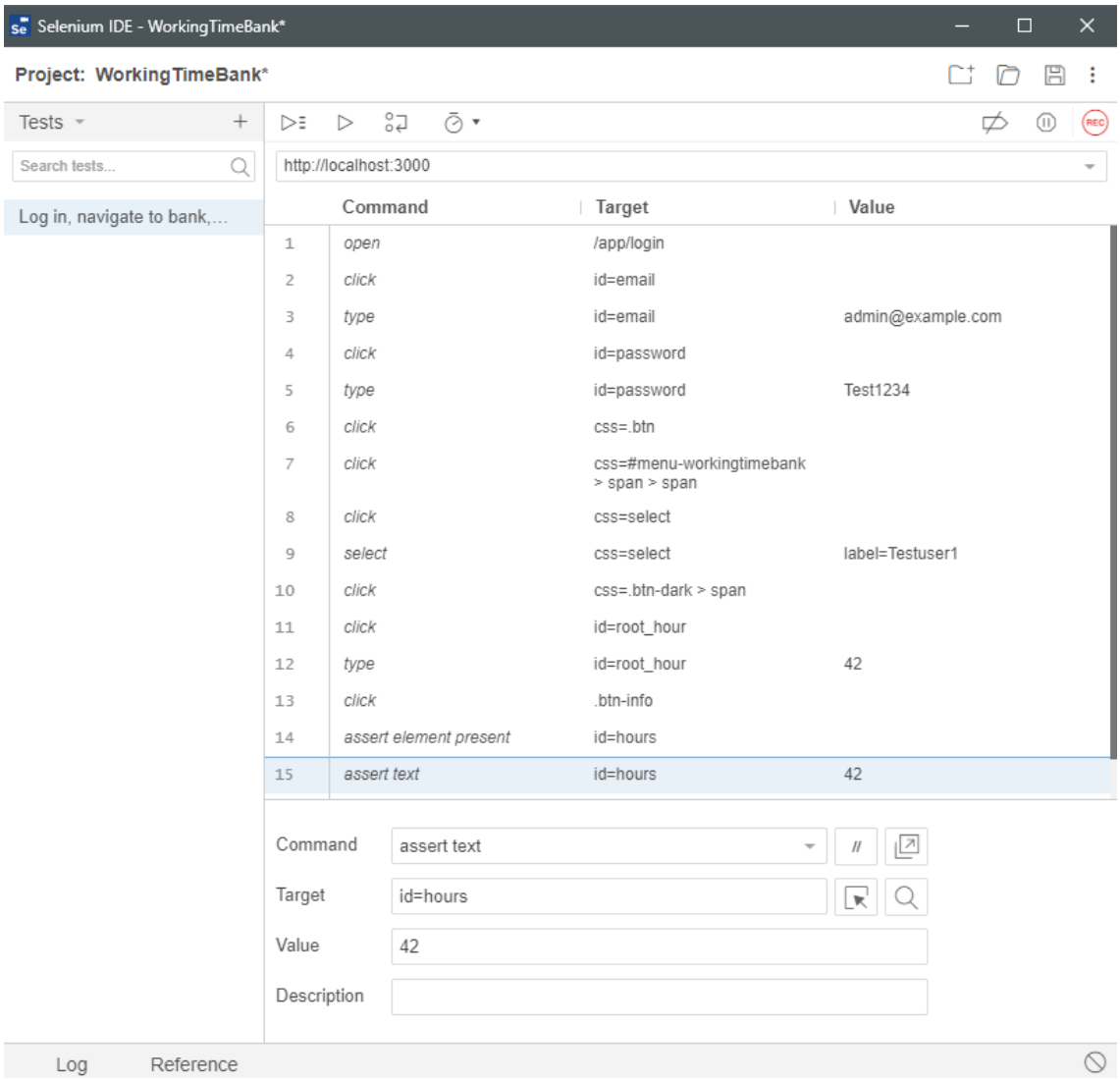
Kun työkalu on asennettu selaimeen ja projekti luotu, on kaikki Selenium IDE:n vaatima konfiguraatio tehty. Seuraavaksi voidaan aloittaa testitapausten luominen nauhoittamalla selaimen toimintaa. Nauhoitus tapahtuu klikkaamalla IDE:n oikeassa ylälaidassa olevaa ”Start recording” -painiketta, jolloin työkalu avaa selaimen ja siirtyy aiemmin määritettyyn kantaosoitteeseen. Tämän jälkeen sivulla navigoidaan ja tehdään testin vaatimat toiminnot. Selenium tallentaa kaikki interaktiot IDE:ssä luodun projektin testitapaukseen reaaliaikaisesti samalla, kun käyttäjä navigoi sivulla (kuvio 33).



Kuvio 33. Selenium IDE nauhoittaa käyttäjän toimintoja selaimessa.

Komentoja voi myös muokata ja lisätä tai poistaa manuaalisesti IDE:n kautta, eikä kaikkea ole pakko tehdä nauhoituksen avulla. Tämä on hyödyllistä etenkin silloin, kun testitapausta haluaa hienosäätää ja yksinkertaistaa, tai jos testiä tehdessä painoi väärää kohtaa eikä koko testiä jaksa nauhoittaa uudelleen. Elementtien hakeminen sivustolta manuaalisesti onnistuu, kun tietää esimerkiksi haettavan elementin id-attribuutin arvon. Vaihtoehtoisesti voi käyttää IDE:n tarjoamaa ”Select target in page” -toimintoa, jonka avulla käyttäjä voi yksinkertaisesti klikata haluamaansa elementtiä sivustolla, jolloin IDE antaa valitun elementin tiedot.

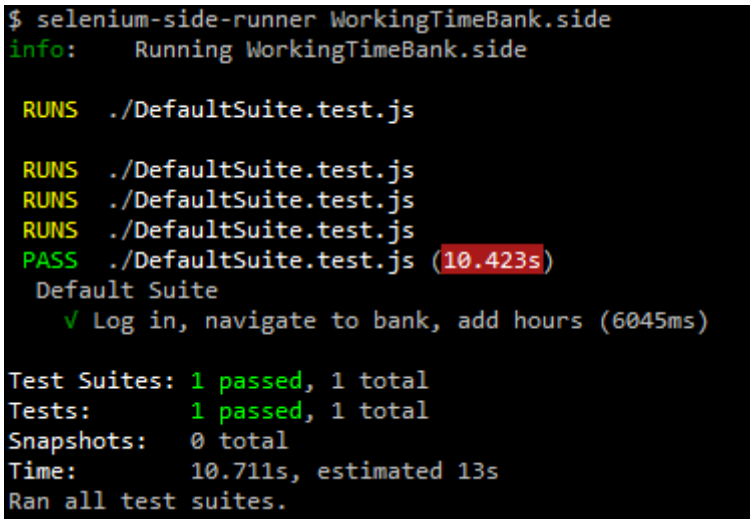
Testitapauksessa oli myös vaatimus, että testi varmistaa syötettyjen tuntien näkymisen sivustolla. Tätä toimintoa ei voida varsinaisesti toteuttaa nauhoittamalla selainta, mutta IDE tarjoaa erilaisia todennuskomentoja tapauksen todentamiseksi. Tässä testitapauksessa varmistetaan, että tunnit-elementti on piirretty sivulle ja sen arvo on sama kuin syötettyjen tuntien arvo. Kun testitapaus on valmis, voidaan se tallentaa myöhempää käyttöä varten. Tallennan luodun testitapauksen, jotta voin suorittaa sen myöhemmin komentorivillä, nimellä WorkingTimeBank.side. Kokonaisuudessaan suunnitelman mukaan luotu testitapaus näyttää Selenium IDE:ssä kuvion 34 mukaiselta.



Kuvio 34. Valmis suunnitelman mukainen testitapaus Selenium IDE:ssä.

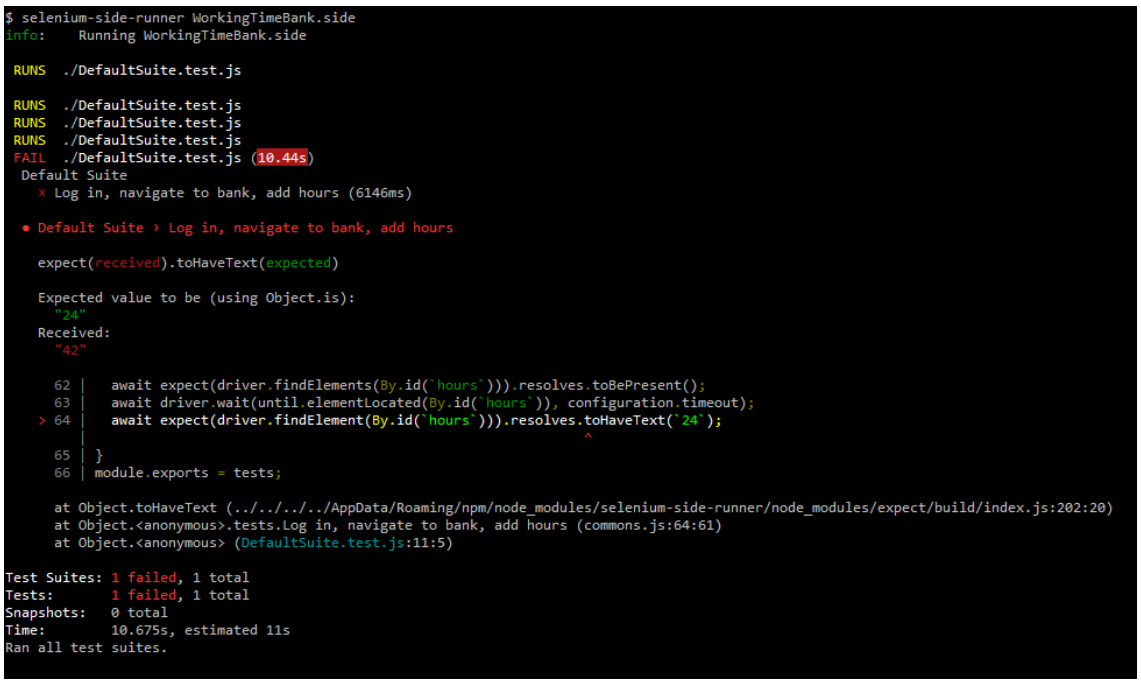
Tämän jälkeen testi voidaan suorittaa joko IDE:ssä valitsemalla ”Run current test” tai ”Run all tests”, tai Selenium side-runner lisäosan avulla komentorivillä. Käytän testin suorittamiseen side-runner komentorivityökalua, sillä se mahdollistaa testien integroimisen CI/CD -putkeen, mikä on jatkokehityksen kannalta tärkeää.

Side-runner vaatii hieman enemmän konfigurointia kuin Selenium IDE. Sitä varten testin suorittavaan laitteeseen tulee asentaa Node.js ja npm, selainajuri (browser driver) sekä itse Selenium side-runner ohjelma. (Selenium 2019c). Kun vaadittavat työkalut on asennettu, voidaan äsken IDE:llä luotu testi ajaa komentoriviltä komennolla ”selenium-side-runner WorkingTimeBank.side” (kuvio 35).



Kuvio 35. Testi suoritetaan onnistuneesti komentorivillä.

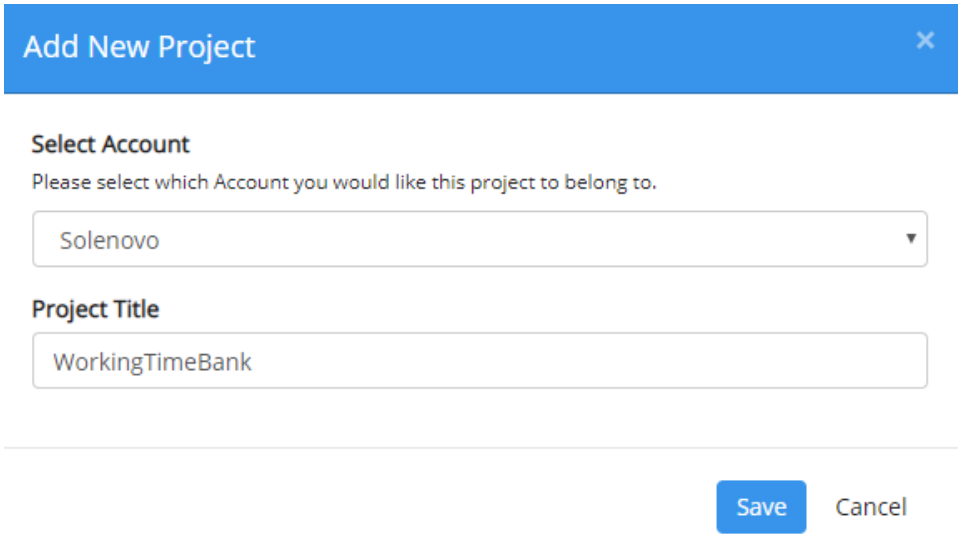
Havainnollistamisen vuoksi kokeillaan vielä saada testi tarkoituksella epäonnistumaan. Vaihdan testitapauksessa tuntien todennuksen vääräksi, luvusta 42 lukuun 24 ja ajan testin uudelleen. Testin suorittaminen epäonnistuu, kuten oli oletettavissa. Selenium ilmoittaa, että testi odotti arvon olevan 24, kun se todellisuudessa olikin 42 (kuvio 36).



Kuvio 36. Testi epäonnistuu, kun arvo muutetaan tarkoituksella vääräksi.

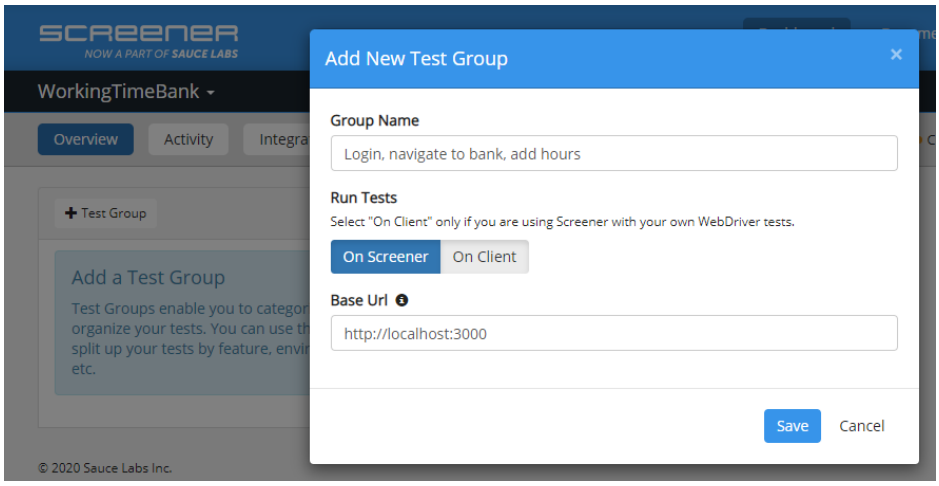
8.3 Käyttöliittymätestin toteuttaminen Screener E2E -työkalulla

Screener E2E -testin luominen aloitetaan luomalla uusi projekti Screener-pilvipalvelussa (kuvio 37). Kun projekti on luotu, voidaan siihen lisätä uusia testitapauksia.



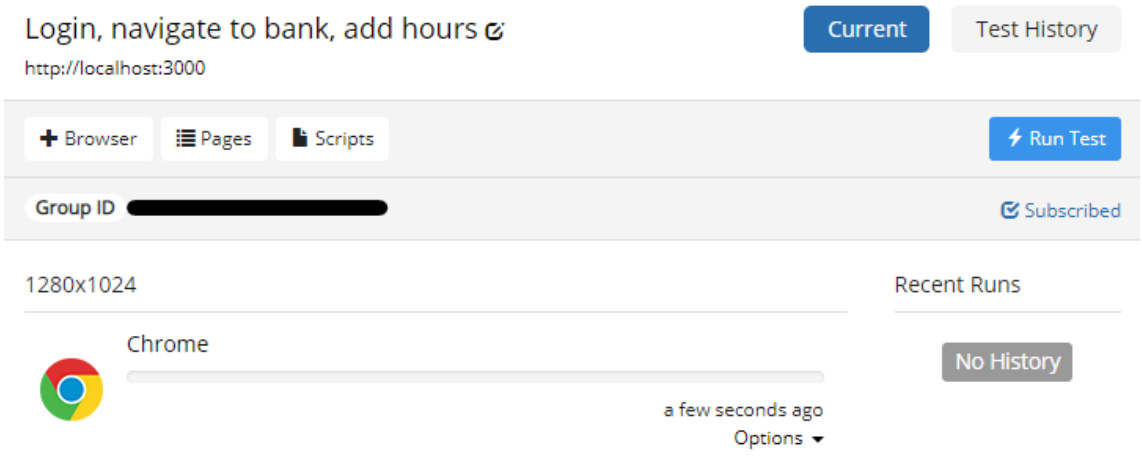
Kuvio 37. Uuden projektin luominen Screener-palvelussa.

Testitapauksen luominen edellyttää käyttäjää syöttämään palveluun testiryhmän nimen ja kantaosoitteen, kuten on kuvattu kuviossa 38. Lisäksi käyttäjä voi valita haluaako hän luoda testit Screenerin pilvipalvelussa, vai integroida olemassa olevia testejä palveluun. Tässä tapauksessa valitaan ensimmäinen vaihtoehto, koska haluan luoda testin kokonaan Screenerin avulla, enkä käyttää äsken luotua Selenium-testiä. Nimeän testiryhmän testitapausta kuvaavalla tavalla ja annan kantaosoitteeksi edelleen osoitteen, jossa sovellus pyörii paikallisesti.



Kuvio 38. Testitapauksen alustaminen Screener-palvelussa.

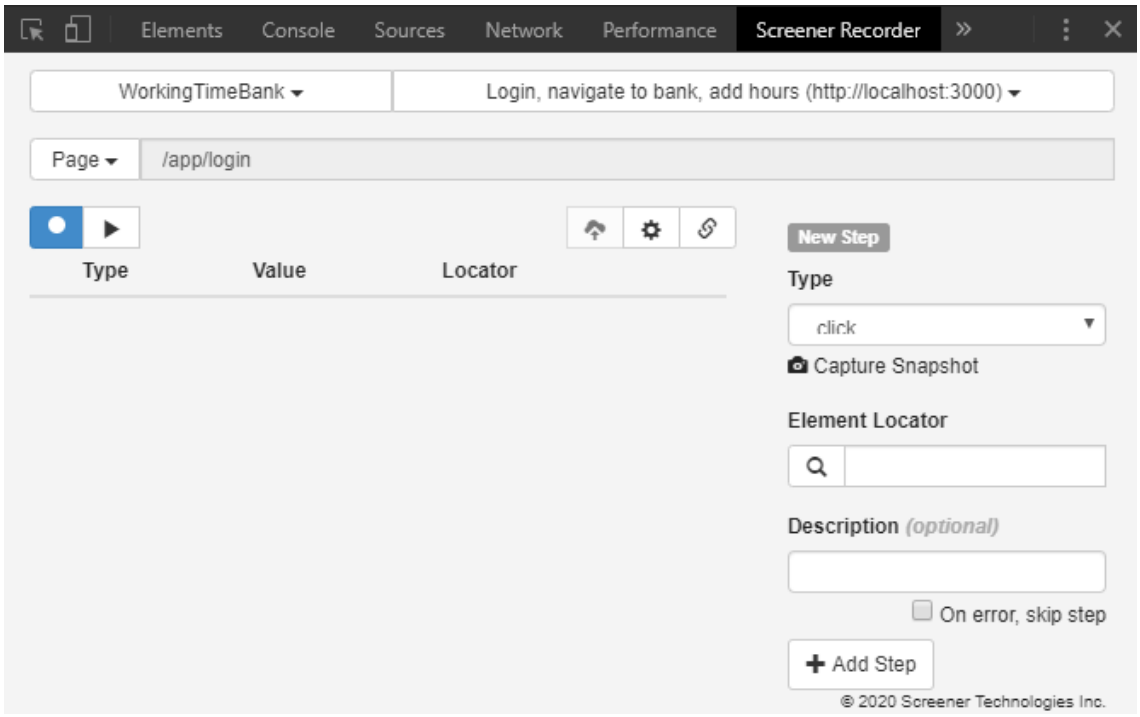
Seuraavaksi testitapausta voidaan määritellä tarkemmin. Testille voidaan asettaa eri vaihtoehdoista tietty selain ja resoluutio, millä testi suoritetaan. Lisäksi testiin voi lisätä sivuja, jotka mahdollistavat testin aloittamisen sovelluksen tietyltä sivulta. Sivut nopeuttavat testien toteuttamista sekä suorittamista, sillä niiden avulla Screenerin ei tarvitse aloittaa jokaista testiä kantaosoitteesta. Asetan testin suoritettavaksi Chrome-selaimella ja lisään siihen sivun ”/app/login”, joka on opinnäytetyön testitapauksessa testin aloitussivu (kuvio 39).



Kuvio 39. Testin suoritusympäristön määrittäminen.

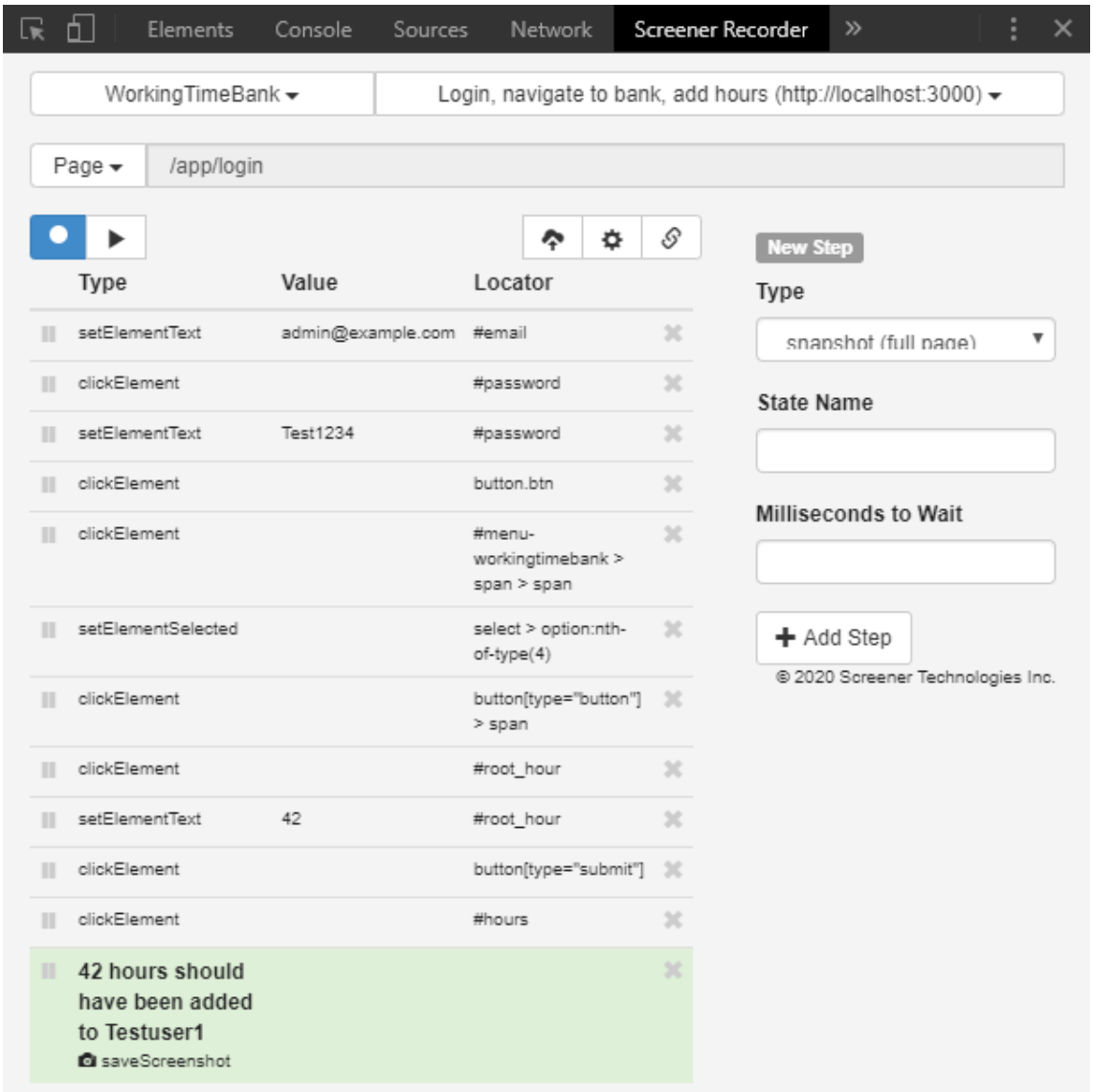
Seuraavaksi luodaan testi. Testin luomista varten täytyy ladata Screener Recorder -lisäosa Chrome-selaimelle. Screener Recorderin käyttöliittymä muistuttaa paljon Selenium IDE:ä. Siinä on painike tallennuksen aloitusta varten, sekä

valikko, josta voi lisätä tai muokata vaiheita manuaalisesti. Käyttöliittymässä näkyy myös äsken luotu projekti, testiryhmä sekä sivu (kuvio 40).



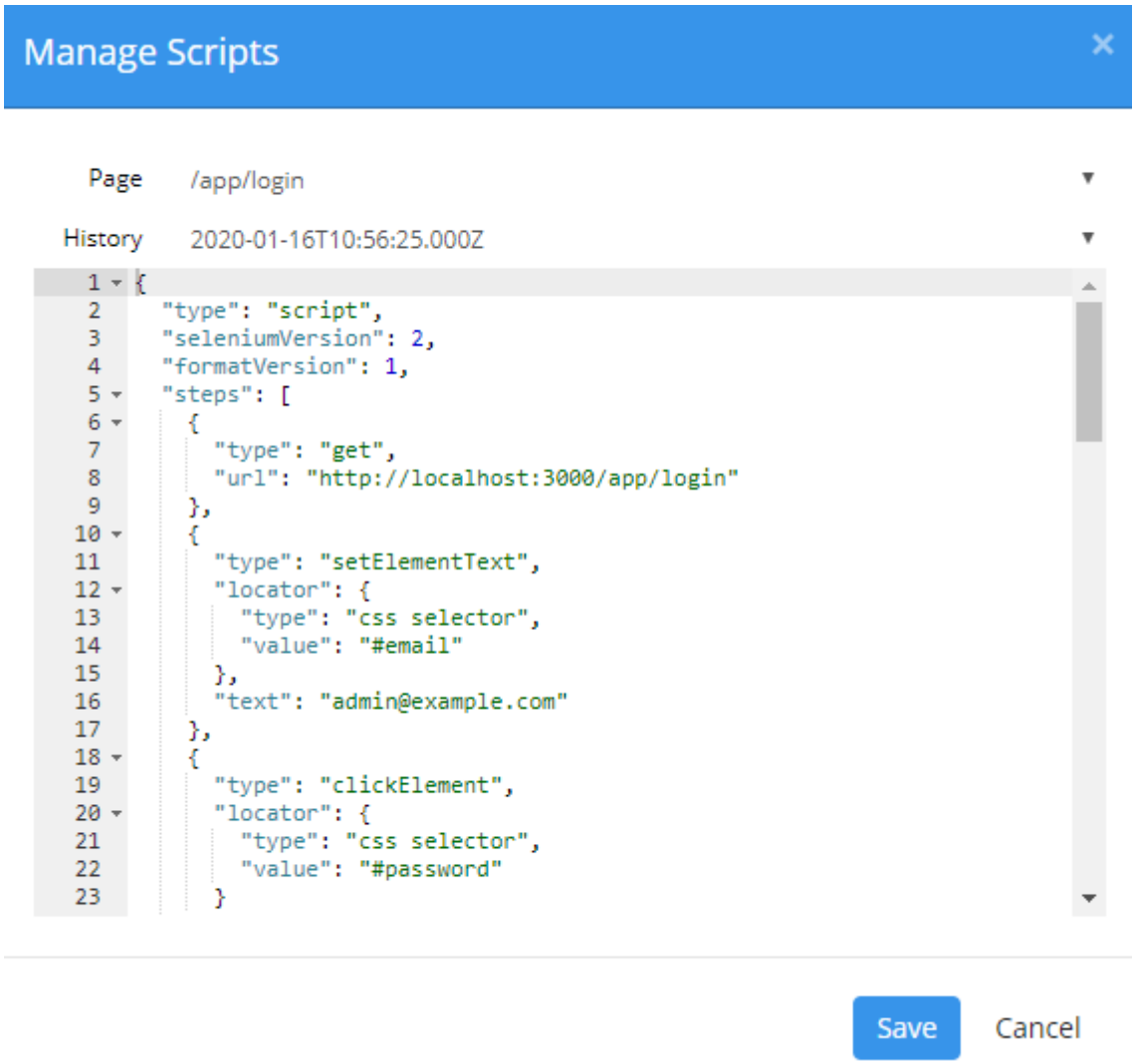
Kuvio 40. Screener Recorderin käyttöliittymä toimii Chromen kehittäjätyökalut-valikossa.

Testin nauhoitus käynnistetään painamalla Record-painiketta. Tämän jälkeen voidaan sovelluksessa tehdä suunnitelman mukaiset toiminnot ja työkalu tallentaa vaiheet ketjuksi. Toimintaperiaate on täysin sama, kuin Selenium IDE:ssä. Screeneristä ei kuitenkaan löydy vastaavia todennuskomentoja, joilla tuntien lisäys voitaisiin todentaa. Tuntien perille menemisen varmistamiseksi lisään ketjuun vielä komennon, joka klikkaa tunnit-elementtiä. Tämän ei pitäisi olla tietenkään mahdollista, mikäli tuntien lisääminen ei ole onnistunut. Tuntien oikeaa määrää tällä tavoin ei kuitenkaan voi varmistaa. Lisäksi haluan käyttää Screenerin kuvakaappaustoimintoa. Siispä lisään loppuun vielä vaiheen, joka ottaa testin loppunäkymästä kuvakaappauksen. Tällä tavoin voidaan varmistua siitä, että tulevaisuudessa testin lopputuloksen muuttava virhe huomataan. Lopullinen testitapaus Screener Recorderissa on kuvattu kuviossa 41.



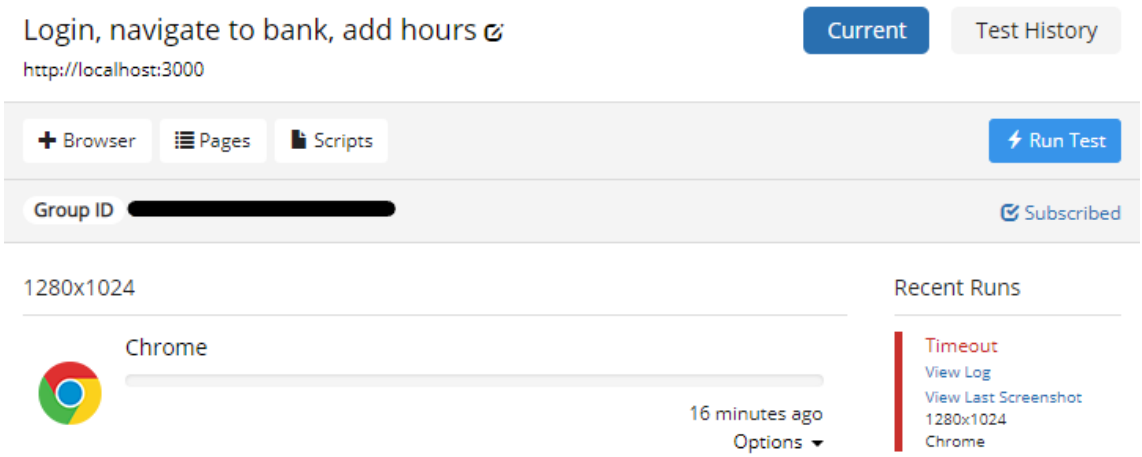
Kuvio 41. Testitapaus nauhoitettuna Screener Recorderiin.

Ennen testin tallentamista Screener-pilvipalveluun voidaan se suorittaa Recorderissa painamalla Play-painiketta. Selain aloittaa automaattisesti suorittamaan ketjun komentoja. Näin varmistutaan siitä, että testi toimii kuten pitääkin ja testiä voidaan vielä helposti korjata tarvittaessa. Kun testi on todettu toimivaksi, tallennetaan se Screener-palveluun valitsemalla ”Save to Cloud”. Testi ilmestyy näkyviin aiemmin palveluun luodun testiryhmän skriptit-osioon, josta testiä voi myös muokata. Muokkaus voi tosin olla tässä vaiheessa hankalaa, sillä kaksitoistavaiheinen testitapaus on kääntynyt palvelussa lähes satariviseksi skriptiksi (kuvio 42).



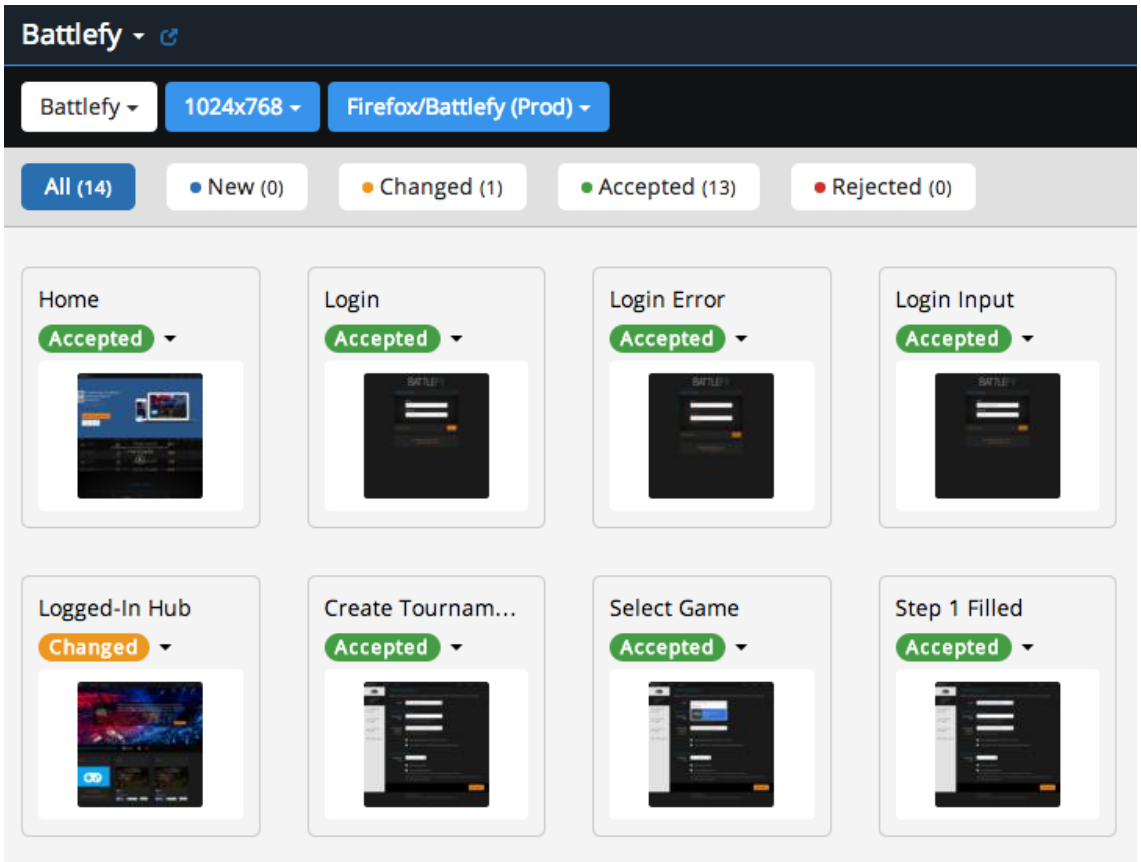
Kuvio 42. Nauhoitettua testiä voi tarkastella ja muokata Screener-palvelussa.

Viimein testi voidaan suorittaa myös Screenerin-pilvipalvelussa valitsemalla ”Run Test”. Palvelu suorittaa testiä hetken ja ilmoittaa sitten, että testi epäonnistui (kuvio 43). Lokitiedostoja tarkastelemalla huomataan, että testi ei ole saanut yhteyttä sivustoon. Tämä on odotettua ja johtuu siitä, että testi yrittää yhdistää localhost-osoitteeseen. Screener ei luonnollisestikaan tätä sivustoa löydä, koska se on pystyssä paikallisesti minun tietokoneellani.



Kuvio 43. Testin suoritus epäonnistuu Screener-pilvipalvelussa.

Jos siis haluaa, että Screenerin E2E testit toimivat kuten on tarkoitus, pitäisi testattavan sivuston olla julkaistu julkiseen osoitteeseen. Toimiessaan oikein, käyttöliittymän kuvakaappauksia vertaileva toiminto antaisi kuvion 44 tyyppisiä tuloksia.



Kuvio 44. Screener E2E havaitsee käyttöliittymässä tapahtuneet muutokset (Screener 2020c).

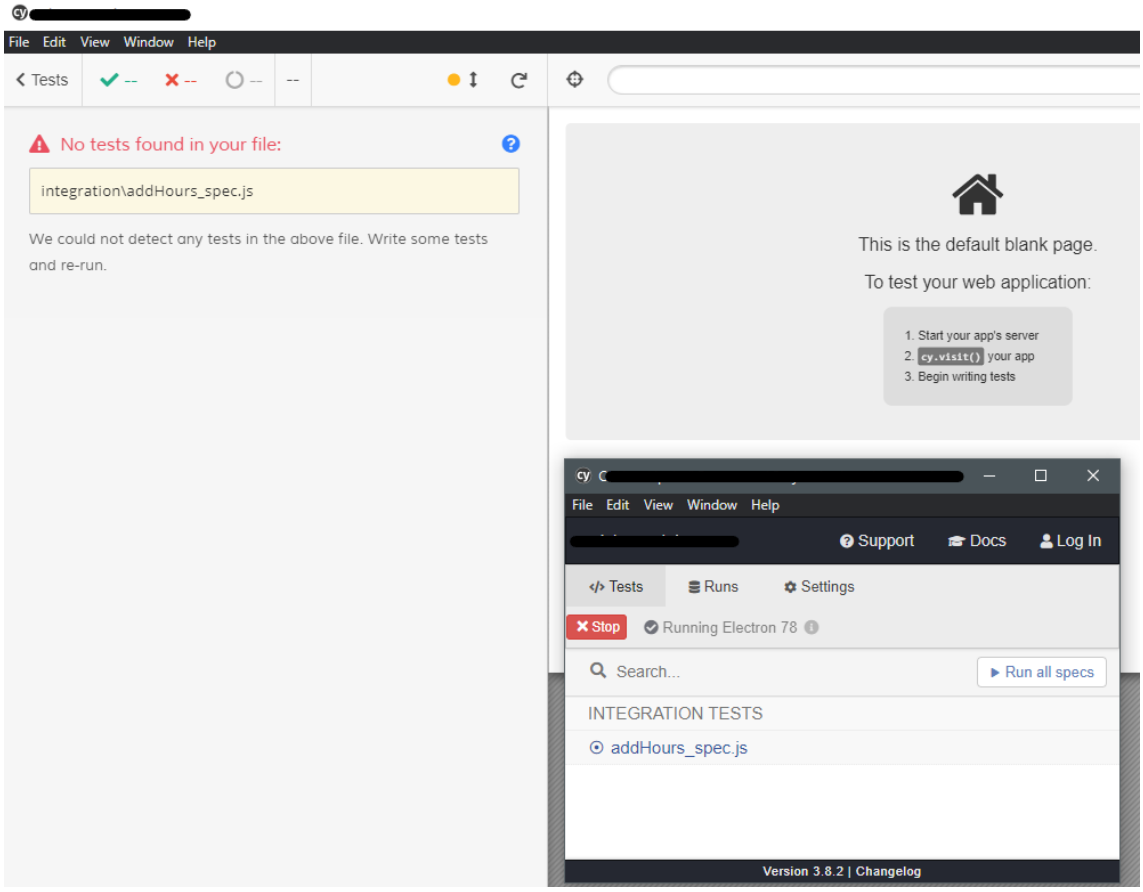
8.4 Käyttöliittymätestin toteuttaminen Cypress-työkalulla

Viimeisenä toteutetaan sama testi vielä Cypressin avulla. Cypress asennetaan testiä suorittavalle laitteelle erillisenä asennustiedostona tai node package managerin avulla ajamalla komentorivillä komento ”npm install cypress” (Cypress 2020c). Tämä vaatii sen, että laitteelle on asennettu Node.js sekä npm tai yarn paketinhallintatyökalu. Kun Cypress on asennettu, voi testitapauksen luomisen aloittaa.

Edellisistä työkaluista poiketen Cypressin testit täytyy kirjoittaa ohjelmakoodiksi nauhoittamisen sijaan. Kun Cypress asennetaan paketinhallintatyökalulla, se luo projektin juureen valmiiksi kansion testitiedostoja varten. Luon siihen kansioon uuden tiedoston, addHours_spec.js, johon kirjoitan testitapauksen. Tiedostoja ei tarvitse nimetä näin, mutta Cypress käyttää omissa ohjeissaan spec-tiedostopäätteitä, ja noudatan samaa käytäntöä.

Cypressin käyttöliittymä aukeaa komentoriviltä komennolla ”npx cypress open”, ja äsken luotu testitiedosto näkyy käyttöliittymässä. Testejä voi suorittaa joko Chrome- tai Electron-selaimessa käyttöliittymän kautta, tai vaihtoehtoisesti komentorivin kautta samoilla selaimilla käyttäen niin sanottua päätöntä (headless) tilaa, joissa itse selainikkuna ei aukea ollenkaan.

Suoritetaan testi ensin käyttöliittymän kautta Electron-selainikkunassa. Kun äsken luotu testi suoritetaan, Cypress avaa selainikkunan, mutta huomauttaa, että tiedostosta ei löytynyt testiä (kuvio 45). Siispä seuraavaksi täytyy aloittaa testin kirjoittaminen.



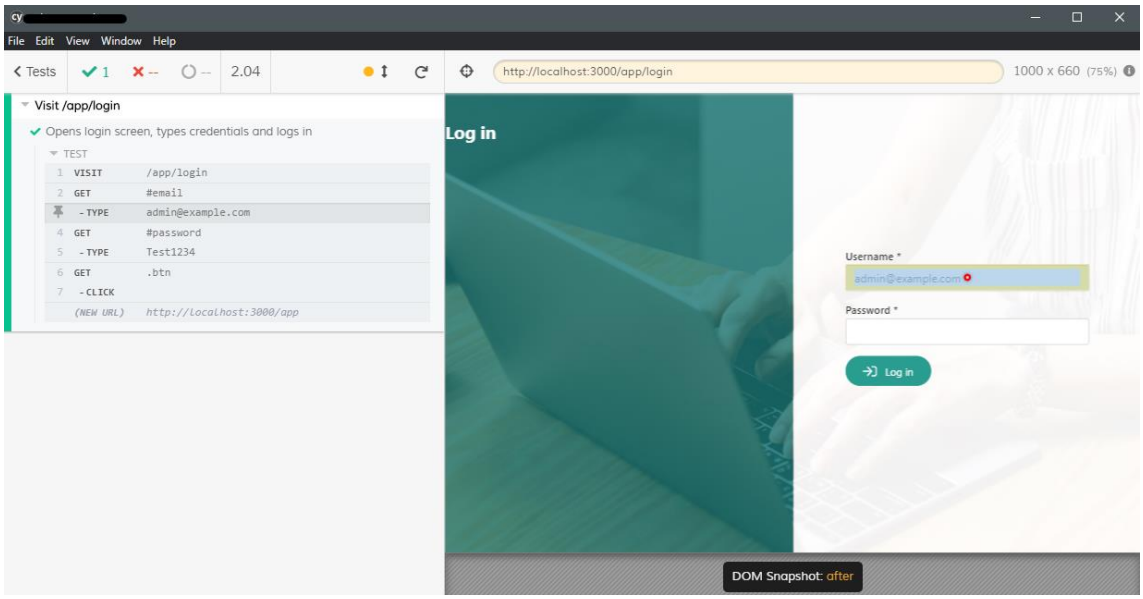
Kuvio 45. Cypress-testin suoritus Electron-selaimessa epäonnistuu, koska tiedosto ei sisällä testiä.

Suorituksen voi jättää taustalle auki samalla, kun testiä kirjoittaa. Selain ja testisuoritus päivittyy reaaliaikaisesti sitä mukaa, kun testitiedostoa muokkaa, joten testin debuggaus on helppoa. Toteutan ensin kokeilumielessä yksinkertaisemman testin, joka avaa kirjautumissivun, kirjoittaa käyttäjätiedot ja klikkaa sisäänkirjautumispainiketta. Tässä vaiheessa testitiedosto näyttää kuvion 46 mukaiselta.

```
// addHours_spec.js
describe('Visit /app/login', () => {
  it('Opens login screen, types credentials and logs in', () => {
    cy.visit('/app/login');
    cy.get('#email').type('admin@example.com');
    cy.get('#password').type('Test1234');
    cy.get('.btn').click();
  });
});
```

Kuvio 46. Ensimmäinen Cypress-testi testaa sisäänkirjautumista.

Cypressin käyttöliittymässä testi suoritetaan onnistuneesti (kuvio 47). Käyttöliittymä mahdollistaa myös jokaisen testin vaiheen erillisen tarkastelun selaimen kanssa. Elementtejä pystyy myös valitsemaan selainikkunasta, joten testissä käytettyjen elementtien nimiä ei tarvitse välttämättä tietää. Sivun aukaisuun riittää tässä tapauksessa pelkkä ”/app/login”, sillä kantaosoite localhost:3000 on määritetty Cypressin konfiguraatiotiedostossa.



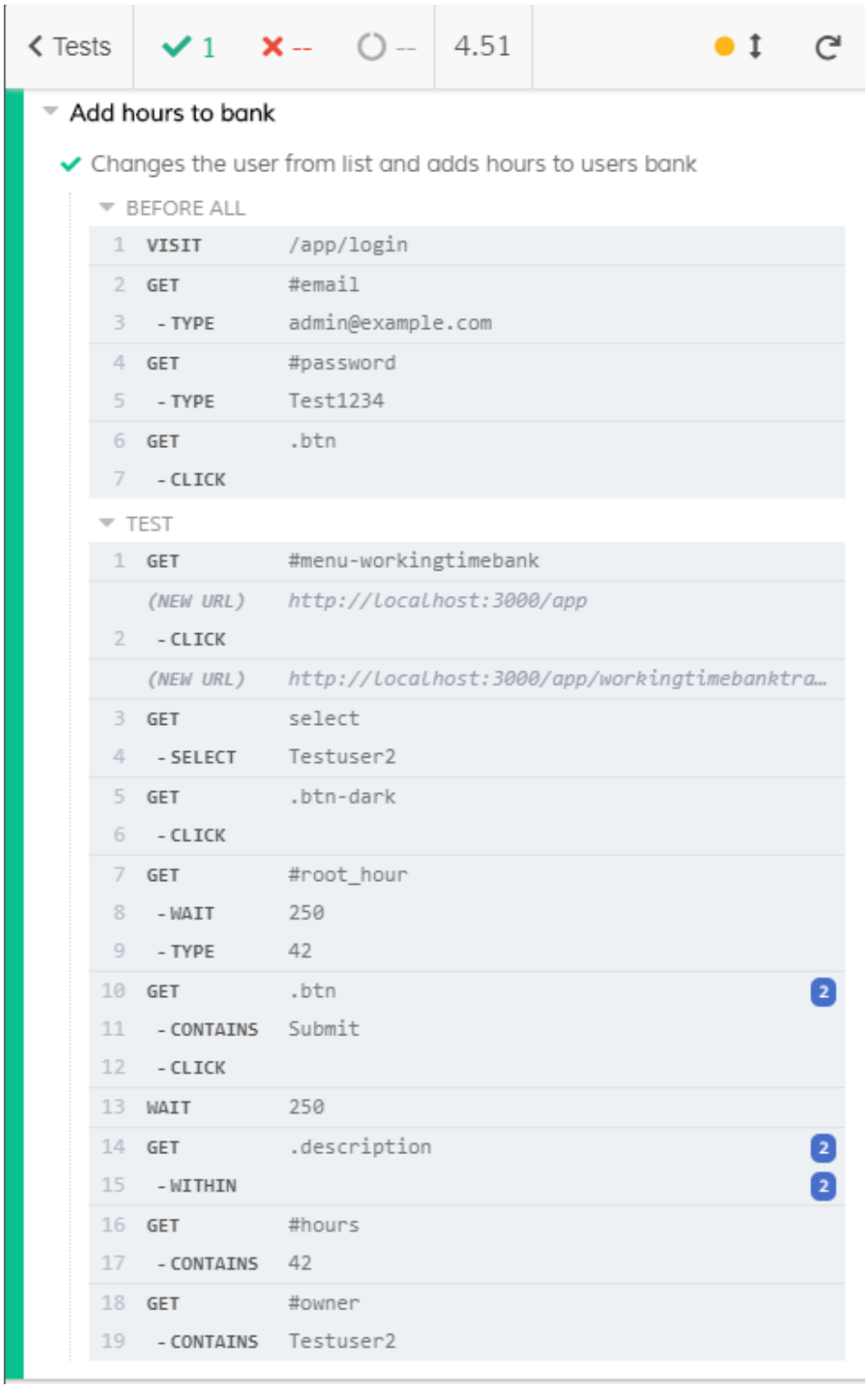
Kuvio 47. Sisäänkirjautuminen Cypressin avulla onnistuu.

Lisään testiin loput suunnitelman mukaisista toiminnoista, ja irrotan kirjautumisen erilliseksi Cypressin tarjoamaksi before-metodiksi. Before mahdollistaa halutun koodinpätkän suorittamisen ennen muita testejä (Cypress 2020d). Käytän sitä tässä sisäänkirjautumiseen, sillä jos testejä halutaan myöhemmin lisätä, pitäisi kirjautuminen sisällyttää erikseen jokaiseen testiin. Testitapauksen mukainen testitiedosto on kuvattu kuviossa 48 ja kuviossa 49 näkyy lopullisen testitapauksen suoritus Cypress-käyttöliittymän avulla.

```
// addHours_spec.js
describe('Add hours to bank', () => {
  before(() => {
    cy.visit('/app/login');
    cy.get('#email').type('admin@example.com');
    cy.get('#password').type('Test1234');
    cy.get('.btn').click();
  });

  it('Navigate to bank, adds hours to user', () => {
    cy.get('#menu-workingtimebank').click();
    cy.get('select').select('Testuser2');
    cy.get('.btn-dark').click();
    cy.get('#root_hour').wait(250).type('42');
    cy.get('.btn').contains('Submit').click();
    cy.wait(250);
    cy.get('.description').within(() => {
      cy.get('#hours').contains('42');
      cy.get('#owner').contains('Testuser2');
    });
  });
});
```

Kuvio 48. Lopullinen suunnitelman mukaan toteutettu Cypress-testi.

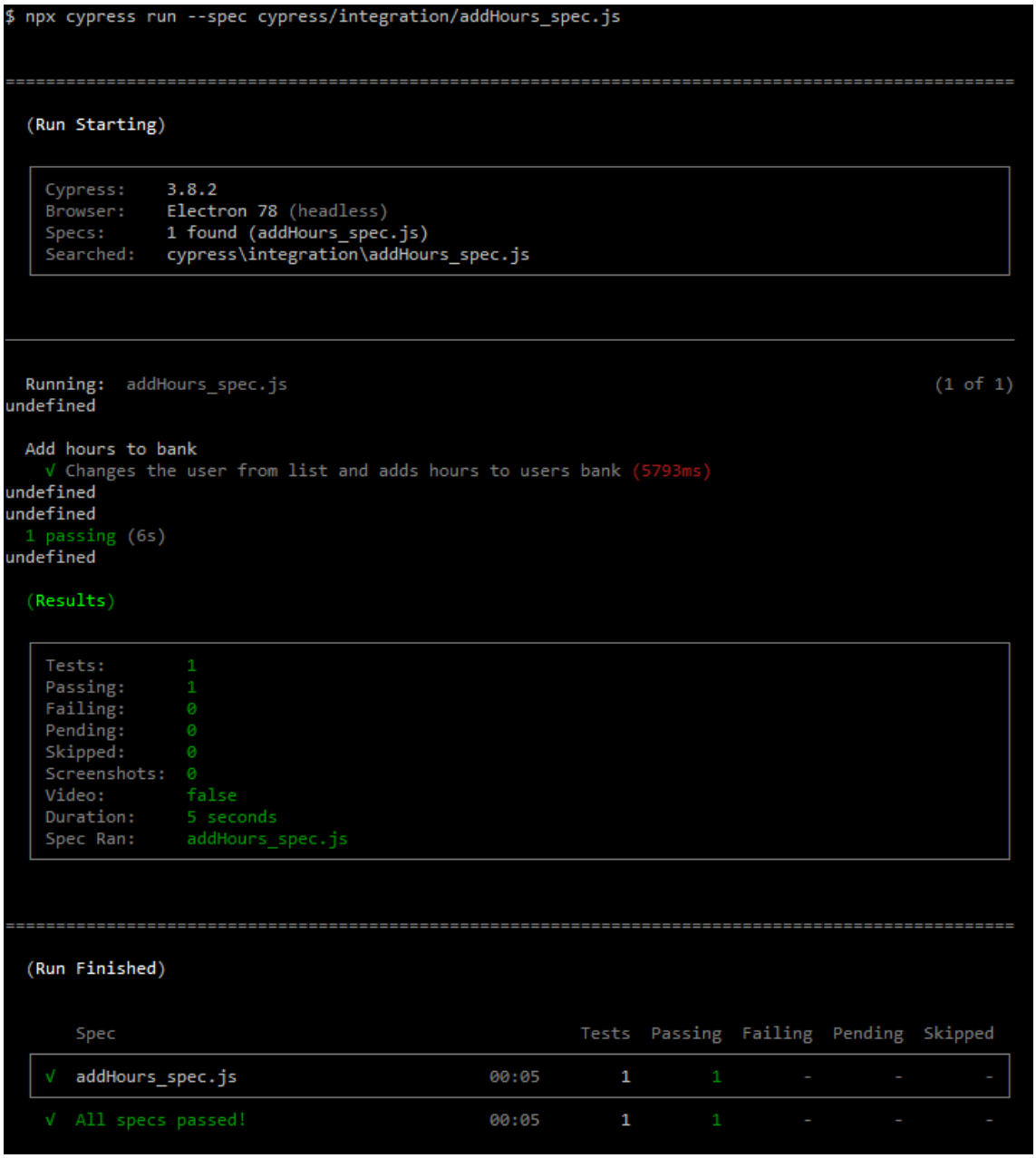


Kuvio 49. Lopullisen testin suorittaminen Cypress-käyttöliittymässä.

Testien kirjoittaminen Cypressillä oli suoraviivaista ja onnistui helposti valikoimalla elementtejä selaimesta käyttöliittymän avulla. Työkalun dokumentaatio oli myös kattava ja helposti ymmärrettävä. Tuntien kirjauksessa oli kuitenkin haasteita, sillä tuntien kirjausikkuna toteuttaa auetessaan ja sulkeutuessaan animaation, eikä Cypress osannut odottaa animaation loppumista. Cypressin pitäisi

osata odottaa elementtien latausta, sillä muuten sillä ei voisi testata dynaamisia sivuja, mutta jostain syystä tämän komponentin animaatiota työkalu ei ymmärtänyt. Tästä syystä testiin piti lisätä wait-metodi, joka odottaa animaation päättymistä sekä ennen tuntien kirjausta, että kirjauksen jälkeen. Wait-metodia käyttämällä testi meni läpi kuten pitikin. Cypressillä tuntien kirjauksen onnistumisen pystyy helposti todentamaan hakemalla tunnit-elementin sisältö ja varmistamalla, että sen arvo on sama kuin kirjattujen tuntien.

Suoritetaan testi vielä komentorivin kautta Electronin-päättömässä tilassa. Tämä tapa mahdollistaa testien integroimisen helposti CI/CD -putkeen. Testi suoritetaan ajamalla komentorivillä komento ”npx cypress run” ja määrittämällä halutessaan tietty testitiedosto spec-lipun (flag) avulla (kuvio 50).



Kuvio 50. Cypress-testin suorittaminen komentorivillä.

9 Tulokset

Tämän opinnäytetyön tarkoitus oli tutkia Javascript-testausta ja siinä käytettäviä erilaisia testauskehyksiä. Javascript-testausosaamista ja -tietoa oli tarkoitus siirtää toimeksiantajan alihankkijalta toimeksiantajalle. Testejä toteutettiin toimeksiantajan työaikapankki-projektissa. Testien ja opinnäytetyön tuoman

testaustietoisuuden vaikutuksia toimeksiantajan tapaan kehittää ohjelmistoja oli tarkoitus mitata ja pohtia.

Ennen raportin kirjoittamisen aloittamista, tutustuessani opinnäytetyön aiheeseen, tein toimeksiantajan projektiin automaatiotestausta. Vuoden vaihteessa automaatiotestit nappasivat kiinni yhden bugin. Tekemäni alustava tutkimustyö esti tämän bugin pääsemisen tuotantokäyttöön. Opinnäytetyö paransi siis tuotteen laatua.

Opinnäytetyön toteutuksen aikana toimeksiantaja sai lisää tietoa erilaisista tavoista testata Javascript-ohjelmistoja. Toimeksiantaja sai myös valmiin esimerkkien kanssa toteutetun raportin erilaisista Javascript-testauskehyksistä. Lisäksi tietoa ja osaamista alihankkijan käyttämistä testaustyökaluista siirtyi järjestetyn koulutuksen ja tutkimuksen myötä minulle ja sitä myöten myös toimeksiantajalle.

Opinnäytetyön aikana suoritetun tutkimuksen ja toiminnallisen osion testauskehyksien vertailu parantaa toimeksiantajan valmiuksia ottaa käyttöön erilaisia testausteknologioita tulevaisuudessa, koska testauskehyksistä on opinnäytetyön myötä yrityksen sisäistä tietoa ja dokumentaatio. Raporttiin päätyneiden testauskehyksien lisäksi tutkimusvaiheessa tutustuin useisiin muihinkin vaihtoehtoihin, joten jatkossa minun on helpompi valita testauskehys eri projekteja ja erilaisia tilanteita varten.

Testit, jotka opinnäytetyön toteutusvaiheessa toteutettiin, tehtiin uuteen projektiin. Tästä syystä testien vaikutuksien mittaaminen testattavan tuotteen kohdalla on hankalaa, ellei mahdotonta. Uudessa projektissa ei ollut olemassa olevia testejä, joten esimerkiksi testikattavuuden kasvua on mahdotonta mitata. Testattavan tuotteen testikattavuus on joka tapauksessa noussut, koska lähtötilanteessa se oli nolla. Tulosten mittaaminen ja analysointi jäi vähäiseksi edellä mainituista syistä ja tuloksia on käsitelty enemmän yrityksessä sisäisesti, kuin tässä raportissa. Tulosten sisäisessä käsittelyssä on keskusteltu siitä, millä tavalla testausta tullaan tekemään jatkossa. Työssä käytetyt testauskehykset on tarkoitus esitellä ja kouluttaa kehitystiimille, jonka jälkeen tiimi voi yhdessä pohtia sitä, mitkä työkalut sopivat heidän käyttöönsä. Opinnäytetyön tuoman uuden tiedon tulosten

odotetaan näkyvän käytännössä vasta pitkällä tähtäimellä. Toimeksiantaja on kuitenkin tyytyväinen opinnäytetyön tuomaan uuteen tietoon, ja opinnäytetyön tuloksien odotetaan vaikuttavan positiivisesti yrityksen tapaan kehittää ohjelmistoja tulevaisuudessa. Opinnäytetyöstä saatua tietoa on tarkoitus hyödyntää tulevissa projekteissa edellä mainittujen koulutustilaisuuksien muodossa.

Opinnäytetyön tuoman tiedon hyödyntämistä ja testien jatkokehitystä on suunniteltu toimeksiantajan kanssa. Tulevaisuudessa opinnäytetyön aikana tutkittuja työkaluja tullaan jalkauttamaan toimeksiantajan projekteihin. Kaikki raporttiin valitut testauskehykset mahdollistavat testien integroimisen osaksi CI/CD -putkea, mikä on tärkeää, että testien suorittaminen voidaan jatkossa automatisoida. Lisäksi hyväksi havaittuja testauskehyksiä ja testausmenetelmiä tullaan kouluttamaan yrityksen kehittäjille. Tarkoitus on järjestää työpajoja, joissa esimerkiksi tietyn testaustyökalun käyttöä koulutetaan käyttäen tätä opinnäytetyötä viitteenä.

10 Pohdinta

Opinnäytetyön aihe oli mielestäni mielenkiintoinen. Työtä aloittaessa minulla ei ollut juuri lainkaan tietoa siitä, miten Javascript-projekteja testataan. Osaamista sekä Node.js- että React.js -teknologioista minulla kuitenkin oli ennestään, joten niiden testauksen tutkiminen oli mielenkiintoinen ja sopivan haastava aihe.

Node.js -testauskehyksistä Mocha ja Jest olivat hyvin samanlaisia. Molemmissa on todella kattava dokumentaatio, ja niillä voi testata lähes mitä tahansa. Jest on näistä kahdesta enemmän yleispätevä, kun taas Mocha on erikoistunut back-end -koodin testaamiseen. Mochan käytöstä rest-rajapintatestaukseen löytyi myös enemmän kolmansien osapuolien ohjeita internetistä. VREST oli käytettävyydeltään todella helppokäyttöinen, eikä sen käyttö välttämättä vaadi ollenkaan ohjelmointiosaamista. Sen toiminta rajoittui kuitenkin pelkästään rest-rajapintojen testaukseen, eikä palvelu tarjonnut läheskään yhtä paljon erilaisia todennusmetodeja, kuin Mochan kanssa käytetty Chai.js tai Jestin kanssa käytetty Expect.js.

React.js -käyttöliittymän testaukseen käytetyistä työkaluista Selenium ja Screener olivat käytettävyydeltään lähes identtisiä, eikä kummankaan käyttö vaadi ohjelmointiosaamista. Molempien työkalujen dokumentaatio oli kuitenkin melko heikko, eikä etenkään maksullisen Screenerin käytöstä löytynyt internetistä juurikaan tietoa. Screener mahdollisti ainoana testatuista työkaluista visuaalisen testauksen suoraan paketista, mutta toiminto vaati toimiakseen sovelluksen julkaisun. Siispä Screenerin visuaalisen testauksen käyttö kehityksen alkuvaiheissa ei ole mahdollista, ellei sovellusta julkaista esimerkiksi testipalvelimelle. Cypressin dokumentaatio oli paras näistä kaikista, ja lisäksi dokumentaatio tarjosi hyviä esimerkkejä ja videoita Cypressin käytöstä. Työkalu on melko uusi, joten apua sen käyttöön kolmansien osapuolien sivustoilta ei löytynyt hirveästi. Tietoa on olemassa kuitenkin sen verran, että pikaisella Google-haulla löytyi vastaus minulle eteen tulleisiin ongelmiin, joihin Cypressin dokumentaatio ei vastannut.

Kaiken kaikkiaan opinnäytetyön tulokset olivat hyviä ja odotettuja. Tulosten mittaamisen haastavuus oli odotettua, koska testit päätettiin toteuttaa uuteen projektiin ja tiesimme etukäteen, että esimerkiksi testauskattavuuden muutosta ei voida uudessa projektissa havaita. Suurin osa hyödyistä näkyy varmasti vasta pitkän ajan kuluttua, mutta testaustietoisuus on kuitenkin yrityksessä kasvanut, mikä heijastuu toivottavasti tuotteen laadun paranemisena. Työn tuoma tieto on ollut hyödyllistä sekä minulle itselleni että toimeksiantajalle. Minulla on nyt paremmat valmiudet toimia työelämässä ohjelmistokehittäjänä ja ymmärtää testauksen merkitystä ja toteuttamista ohjelmistokehitysprojekteissa. Toimeksiantaja taas sai siirrettyä osaamista alihankkijalta yrityksen sisälle sekä kerättyä tietoa Javascript-testauskehysten toiminnasta ja niiden käytöstä. Itse raportin kirjoittaminen oli kohtalaisen helppoa, eikä suurempia ongelmia tai esteitä työn aikana ilmentynyt. Ongelmatilanteissa sain aina apua joko toimeksiantajalta tai opinnäytetyön ohjaajalta. Lopuksi haluaisin kiittää vielä toimeksiantajaa ja oppilaitosta sekä kaikkia, jotka ovat jollain tapaa olleet tukenani tämän opinnäytetyöprosessin aikana.

Lähteet

Arnold, D. 2000. The Explosion of the Ariane 5. University of Minnesota. http://www-users.math.umn.edu/~arnold/disasters/ariane.html. 26.11.2019.

Capan, T. 2013. Why The Hell Would I Use Node.js? A Case-by-Case Tutorial. Toptal. https://www.toptal.com/nodejs/why-the-hell-would-i-use-node-js. 13.12.2019.

Codecademy. 2019. React: The Virtual DOM. https://www.codecademy.com/articles/react-virtual-dom. 17.12.2019.

Cuomo, J. 2013. JavaScript Everywhere and the Three Amigos. IBM. Internet Archive. https://web.archive.org/web/20191106190801/https://www.ibm.com/developerworks/community/blogs/gcuomo/entry/javascript_everywhere_and_the_three_amigos. 27.1.2020.

Cypress. 2020a. Cypress Pricing Plans. https://www.cypress.io/pricing. 14.1.2020.

Cypress. 2020b. Why Cypress? https://docs.cypress.io/guides/overview/why-cypress.html. 10.1.2020.

Cypress. 2020c. Installing Cypress. https://docs.cypress.io/guides/getting-started/installing-cypress.html. 16.1.2020.

Cypress. 2020d. Writing and Organizing Tests: Hooks. https://docs.cypress.io/guides/core-concepts/writing-and-organizing-tests.html#Hooks. 16.1.2020.

Dahl, R. 2010. Joyent & Node. Google Groups. https://groups.google.com/forum/#!topic/nodejs/lWo0MbHZ6Tc. 10.12.2019.

Dahl, R. 2013. New gatekeeper. Google Groups. https://groups.google.com/forum/#!topic/nodejs/hfajgpvGTLY. 10.12.2019.

Dash Magazine. 2019. Codeburst.io. https://codeburst.io/14-frequently-asked-questions-about-reactjs-68fd29e8dfea. 16.12.2019.

Deb, S. 2019. Software Testing Life Cycle- Different Stages of Testing. Edureka. https://www.edureka.co/blog/software-testing-life-cycle. 29.11.2019.

Deno. 2019. Deno Manual. https://deno.land/std/manual.md. 10.12.2019.

Elliot, B. 2016. JavaScript Testing: Unit vs Functional vs Integration Tests. Sitepoint. https://www.sitepoint.com/javascript-testing-unit-functional-integration.

Ericsson. 2019. Ericsson Mobility Report. https://www.ericsson.com/en/mobility-report/reports/november-2019. 26.11.2019

Eriksson, U. 2014a. Differences Between the Different Levels of Testing. Reqtest. https://reqtest.com/testing-blog/different-levels-of-testing. 2.12.2019.

Eriksson, U. 2014b. A guide to excellent Acceptance Testing. Reqtest. https://reqtest.com/testing-blog/a-guide-to-excellent-acceptance-testing. 2.12.2019.

Eriksson, U. 2016. 7 Things About Software System Testing You Should Know. Reqtest. https://reqtest.com/testing-blog/software-system-testing. 2.12.2019.

Everett G. & McLeod R. 2007. Software Testing: Testing Across the Entire Software Development Lifecycle. Hoboken, New Jersey: John Wiley & Sons, Inc.

Facebook. 2019a. React. GitHub. https://github.com/facebook/react. 16.12.2019.

Facebook. 2019b. Virtual DOM and Internals. React. https://reactjs.org/docs/faq-internals.html. 17.12.2019.

Facebook. 2020a. Jest. GitHub. https://github.com/facebook/jest. 17.1.2020.

Facebook. 2020b. Jest: Delightful JavaScript Testing Framework. https://jestjs.io. 17.1.2020.

Facebook. 2020c. Jest: Expect. https://jestjs.io/docs/en/expect. 22.1.2020.

Facebook. 2020d. Jest: Testing Asynchronous Code. https://jestjs.io/docs/en/asynchronous. 22.1.2020.

Gelles, D. 2019. Boeing 737 Max: What's Happened After the 2 Deadly Crashes. The New York Times. https://www.nytimes.com/interactive/2019/business/boeing-737-crashes.html. 26.11.2019.

Greif, S., Benitte R. & Rambeau, M. 2018a. The State of Javascript 2018: Testing - Conclusion. StateOfJS. https://2018.stateofjs.com/testing/conclusion. 3.12.2019.

Greif, S., Benitte R. & Rambeau, M. 2018b. The State of Javascript 2018: Testing - Jest. StateOfJS. https://2018.stateofjs.com/testing/jest. 3.12.2019.

Greif, S., Benitte R. & Rambeau, M. 2018c. The State of Javascript 2018: Testing - Overview. StateOfJS. https://2018.stateofjs.com/testing/overview. 3.12.2019.

Guru99. 2020a. What is V Model in Software Testing? Learn with SDLC & STLC Example. https://www.guru99.com/v-model-software-testing.html. 14.1.2020.

Guru99. 2020b. What is Agile Testing? Process, Strategy, Test Plan, Life Cycle Example. https://www.guru99.com/agile-testing-a-beginner-s-guide.html. 14.1.2020.

Guru99. 2020c. STLC - Software Testing Life Cycle Phases & Entry, Exit Criteria. https://www.guru99.com/software-testing-life-cycle.html. 14.1.2020.

Guru99. 2020d. Test Environmet for Software Testing. https://www.guru99.com/test-environment-software-testing.html. 14.1.2020.

Hamedani, M. 2018. React Virtual DOM Explained in Simple English. Programming with Mosh. https://programmingwithmosh.com/react/react-virtual-dom-explained. 17.12.2019.

Heller, M. 2017. What is Node.js? InfoWorld. The JavaScript runtime explained. https://www.infoworld.com/article/3210589/what-is-nodejs-javascript-runtime-explained.html. 13.12.2019.

Herrin, J. 2019. Internet of Things (IoT) trends in 2019. Infosec. https://resources.infosecinstitute.com/internet-of-things-iot-trends. 3.12.2019.

Jones, C. 2012. A Short History of Cost Per Defect Metric. David Consulting Group. https://www.softwarevalue.com/media/389295/cost-per-defect-2013.pdf. 28.11.2019.

JSConf. 2013. Ryan Dahl: Node JS. Youtube. https://www.youtube.com/watch?v=EeYvFl7li9E. 10.12.2019.

JSConf. 2018. 10 Things I Regret About Node.js - Ryan Dahl - JSConf EU. https://www.youtube.com/watch?v=M3BM9TB-8yA. 10.12.2019.
Kasurinen, J.P. 2013. Ohjelmistotestauksen käsikirja. Jyväskylä: Docendo.
Khan Academy. 2019. What's a JS library? https://www.khanacademy.org/computing/computer-programming/html-css-js/using-js-libraries-in-your-webpage/a/whats-a-js-library. 3.12.2019.
Krotoff, T. 2019. Front end frameworks popularity (React, Vue and Angular). GitHub. https://gist.github.com/tkrotoff/b1caa4c3a185629299ec234d2314e190. 16.12.2019.
McCarthy, K. 2011. Node.js Interview: 4 Questions with Creator Ryan Dahl. AmericanInno. https://www.americaninno.com/boston/node-js-interview-4-questions-with-creator-ryan-dahl. 10.12.2019.
MDN Web Docs. 2019. JavaScript. https://developer.mozilla.org/en-US/docs/Web/JavaScript. 9.12.2019.
MDN Web Docs. 2020. General asynchronous programming concepts. https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Asynchronous/Concepts. 27.1.2020.
Mikkonen, J. 2017. Rest on nettipalveluiden yhteinen kieli. Tivi. https://www.tivi.fi/uutiset/rest-on-nettipalveluiden-yhteinen-kieli/23703ab5-dd19-383e-a422-ebfc3d910583. 22.1.2019.
Mocha. 2020a. Mocha - the fun, simple, flexible JavaScript test framework. https://mochajs.org. 17.1.2020.
Mocha. 2020b. Mocha - asynchronous code. https://mochajs.org/#asynchronous-code. 22.1.2020.
NodeJS. 2020a. Overview of Blocking vs Non-Blocking. https://nodejs.org/en/docs/guides/blocking-vs-non-blocking. 29.1.2020.
NodeJS. 2020b. The Node.js Event Loop, Timers, and process.nextTick(). https://nodejs.org/uk/docs/guides/event-loop-timers-and-nexttick. 29.1.2020.
Npm-stat. 2019. Download statistics for package react, vue. https://npm-stat.com/charts.html?package=react&package=vue&from=2013-01-01&to=2019-12-16. 16.12.2019.
Npm-stat. 2020. Download statistics for packages chai, expect, should, better-assert, unexpected. https://npm-stat.com/charts.html?package=chai&package=expect&package=should&package=better-assert&package=unexpected&from=2015-01-01&to=2019-12-31. 17.1.2020.
Politis, D. 2017. The 2017 State of the SaaS-Powered Workplace Report. Bettercloud. https://www.bettercloud.com/monitor/state-of-the-saas-powered-workplace-report. 3.12.2019.
Pressman, R.S. 2010. Software Engineering: A Practioner's Approach. New York: McGraw-Hill.
Reqtest. 2018. Agile Testing - Principles, methods & advantages. https://reqtest.com/testing-blog/agile-testing-principles-methods-advantages. 17.12.2019.
Sampson, B. 2018. Getting to grips with software testing. Aerospace Testing International. https://www.aerospacetestinginternational.com/features/getting-to-grips-with-software-testing.html. 3.12.2019.
Screener. 2020a. Screener.io: Automated Visual Testing. https://screener.io. 14.1.2020.

Screener. 2020b. Screener Recorder. https://screener.io/docs/recorder. 14.1.2020.
Screener. 2020c. Screener Workflow. https://screener.io/docs/workflow. 14.1.2020.
Screener. 2020d. Continuous Integration with Screener. https://screener.io/docs/continuous-integration. 16.1.2020.
Selenium. 2019a. Selenium IDE. https://selenium.dev/selenium-ide 14.1.2020.
Selenium. 2019b. Grid. https://selenium.dev/documentation/en/grid. 14.1.2020.
Selenium. 2019c. Command-line Runner. https://selenium.dev/selenium-ide/docs/en/introduction/command-line-runner. 14.1.2020.
Selenium. 2020a. Quick Tour. https://selenium.dev/documentation/en/getting_started/quick. 14.1.2020.
Selenium. 2020b. WebDriver. https://selenium.dev/documentation/en/webdriver. 14.1.2020.
Shashidhara, P. 2017. Episode 8: Interview with Ryan Dahl, Creator of Node.js. Mapping the Journey. https://mappingthejourney.com/single-post/2017/08/31/episode-8-interview-with-ryan-dahl-creator-of-nodejs. 10.12.2019.
Software Testing Help. 2019a. What Is Software Testing Life Cycle (STLC)? https://www.softwaretestinghelp.com/what-is-software-testing-life-cycle-stlc. 2.12.2019.
Software Testing Help. 2019b. What Is End To End Testing: E2E Testing Framework With Examples. https://www.softwaretesting-help.com/what-is-end-to-end-testing. 16.1.2020.
Stack Overflow. 2016. Stack Overflow Developer Survey 2016 - IV. Trending Tech on Stack Overflow. https://insights.stackoverflow.com/survey/2016#technology-trending-tech-on-stack-overflow. 16.12.2019.
Stack Overflow. 2019. Stack Overflow Developer Survey 2019 - Most Popular Technologies. https://insights.stackoverflow.com/survey/2019#most-popular-technologies. 9.12.2019.
Stackshare. 2019. React. https://stackshare.io/react. 16.12.2019.
Synopsys. 2019. What is continuous testing? https://www.synopsys.com/blogs/software-security/continuous-testing-cicd. 17.1.2020.
Taina, J. 2013. Ohjelmistotuotanto -Luentokalvot. 4. Vaatimusanalyysi. Helsingin Yliopisto. https://www.cs.helsinki.fi/u/taina/ohtu/s-2003/luennot/luku04.pdf. 28.11.2019.
Tieturi. 2006. Testauksen valmennusohjelma v6.4.
Tuovinen, A. 2013. Ohjelmistotestauksen perusteita I. Helsingin Yliopisto. https://www.cs.helsinki.fi/u/aptuovin/testaus/Ohj_testaus_2013_1.pdf. 27.11.2019.
Työaikalaki 872/2019.
Vala Group. 2019. Q&A: Miksi ohjelmistotestaus on tärkeää? https://www.valagroup.com/2019/01/qa-miksi-ohjelmistotestaus-on-tarkeaa. 27.11.2019.
Visionmedia. 2019. Supertest. GitHub. https://github.com/visionmedia/supertest. 22.1.2019.
Vocke, H. 2018. The Practical Test Pyramid. Martinfowler.com. https://martinfowler.com/articles/practical-test-pyramid.html. 10.1.2020.
VREST. 2020a. vREST - Automated REST API Testing Tool. https://vrest.io. 17.1.2020.

VREST. 2020b. vREST - Pricing. https://vrest.io/pricing. 17.1.2020.
W3Schools. 2019. JavaScript HTML DOM. https://www.w3schools.com/js/js_htmldom.asp. 17.12.2019.
Zaidman, V. 2019. An Overview of JavaScript Testing in 2019. https://medium.com/welldone-software/an-overview-of-javascript-testing-in-2019-264e19514d0a. 3.12.2019.